**ISO TC184/SC4/*WG 5 N 213 (P 3 )**
*Complete with EC (for Editing), PMAG, or WG

**Date:** 1 December 1994 **Supercedes SC4/WG 5** N 40 (P )

**PRODUCT DATA REPRESENTATION AND EXCHANGE**

**Part:** 12 **Title:** EXPRESS-I Language Reference Manual

Purpose of this document as it relates to the target document is:

X Primary Content
___ Issue Discussion
___ Alternate Proposal
___ Partial Content

Current Status: Editorially complete

**ABSTRACT:**
The EXPRESS-I instance language provides a means of displaying example instantiations of EXPRESS defined elements and also provides formal support for the specification of test cases.

**KEYWORDS:**
EXPRESS, Instantiation Language, Test Cases

**Document Status/Dates** (dd/mm/yy)

| Part Documents | | Other SC4 Documents | |
|---|---|---|---|
| 2/11/92 | Released | | Released |
| 19/5/94 | Project | | Working |
| 19/5/94 | Working | | Editorial OK |
| 17/10/94 | Technically Complete | | Technically Complete |
| 1/12/94 | Editorially Complete | | Approved |
| | ISO Committee Draft | | |

**Owner/Editor:** Peter R Wilson
**Address:** NIST
Bldg. 220, Room A127
Gaithersburg, MD 20899
USA

**Telephone/FAX:** +1 (301) 975-2976
**E-mail:** `pwilson@cme.nist.gov`

**Alternate:** Philip Spiby
**Address:** CADDETC
Arndale House
Headingley
Leeds, LS6 2UU
United Kingdom

**Telephone/FAX:** +44 532 305005
**E-mail:**

**Comments to Reader**
Members of WG6 have used technically the same version of EXPRESS-I to develop trial Abstract Test Cases. The changes made in this version in preperation for CD balloting are essentially restricted to documentation updates to match the ISO 10303-11:1994 International Standard document.

NOTE: This page is given for information only. It is not part of ISO10303:Part 12.

## Credits

The authors of Express-I are:

| | |
|---|---|
| Paul Bell | CADDETC |
| Ray Goult | LMR Systems |
| Steve Clark | National Institute of Standards and Technology |
| András Márkus | Hungarian Academy of Sciences |
| Sabine Müllenbach | Digital Equipment International GmbH |
| Jon Owen | CADDETC |
| Douglas Schenck | McDonnell Aircraft Company |
| Philip Spiby | CADDETC |
| Bernd Wenzel | Digital Equipment Corp. |
| Peter Wilson | Rensselaer Polytechnic Institute and NIST |

## Meetings

| | |
|---|---|
| San Diego, USA | April 1991 |
| Sapporo, Japan | July 1991 |
| London, UK | July 1992 |
| Dallas, USA | October 1992 |
| Davos, Switzerland | May 1994 |
| Greenville, USA | October 1994 |

## Contents

## Tables

# Foreword

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303–12 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration,* Subcommittee SC4, *Industrial data and global manufacturing programming languages.*

This part of ISO 10303 is based in part upon material in:

- ISO 6937: Information Processing — Coded character sets for text communication.
- ISO TR 9007: Information Processing Systems — Concepts and terminology for the conceptual schema and the information base.
- ISO 10303-11: Product Data Representation and Exchange — Description methods: The EXPRESS language reference manual.

ISO 10303 consists of the following parts under the general title *Industrial automation systems and integration – Product data representation and exchange:*

- Part 1, Overview and fundamental principles;
- Part 11, Description methods: The *EXPRESS* language reference manual;
- Part 12, Description methods: The *EXPRESS-I* language reference manual;
- Part 21, Implementation methods: Clear text encoding of the exchange structure;
- Part 22, Implementation methods: Standard data access interface specification;
- Part 31, Conformance testing methodology and framework: General concepts;
- Part 32, Conformance testing methodology and framework: Requirements on testing laboratories and clients;
- Part 41, Integrated generic resources: Fundamentals of product description and support;
- Part 42, Integrated generic resources: Geometric and topological structures;

- Part 43, Integrated generic resources: Representation structures;
- Part 44, Integrated generic resources: Product structure configuration;
- Part 45, Integrated generic resources: Materials;
- Part 46, Integrated generic resources: Visual presentation;
- Part 47, Integrated generic resources: Shape variation tolerances;
- Part 49, Integrated generic resources: Process structure and properties;
- Part 101, Integrated application resources: Draughting;
- Part 104, Integrated application resources: Finite element analysis;
- Part 105, Integrated application resources: Kinematics;
- Part 201, Application protocol: Explicit draughting;
- Part 202, Application protocol: Associative draughting;
- Part 203, Application protocol: Configuration controled design;
- Part 207, Application protocol: Sheet metal die planning and assembly;
- Part 210, Application protocol: Printed circuit assembly product design data;
- Part 213, Application protocol: Numerical control process plans for machined parts.

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of this International Standard reflects its structure:

- Parts 11 and 12 specify the description methods;
- Parts 21 and 22 specify the implementation methods;
- Parts 31 and 32 specify the conformance testing methodology and framework;
- Parts 41 to 49 specify the integrated generic resources;
- Parts 101 to 105 specify the integrated application resources;
- Parts 201 to 213 specify the application protocols.

Should further parts be published, they will follow the same numbering pattern.

Annexes A, B and C are an integral part of this part of ISO 10303. Annexes D, E, F and G are for information only.

# Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application protocols, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303–1. This part of ISO 10303 is a member of the descriptive methods series.

This part of ISO 10303 specifies the elements of the *EXPRESS-I* language. Each element of the language is presented in its own context with examples. Simple elements are introduced first, then more complex ideas are presented in an incremental manner.

# Language Overview

*EXPRESS-I* is the name of a formal data representation and abstract test case specification language. It may be used to exemplify the information requirements of other parts of this International Standard and is a companion to the *EXPRESS* and *EXPRESS-G* languages. It is based on a number of design goals among which are:

– The size and complexity of ISO 10303 demands that the language be parsable by both computers and humans. Expressing elements of ISO 10303 in a less formal manner would eliminate the possibility of employing computer automation in checking for inconsistencies in presentation or specification.

– The language focuses on the display of the realisation of the properties of entities, which represent objects of interest. The definition of an entity is in terms of its properties, which are charcterized by specification of a domain and the constraints on that domain.

– The language seeks to avoid, as far as possible, specific implementation views.

– Provision of a means of displaying small populated models of *EXPRESS* schemas.

– Provision of a means of supporting the specification of test suites for information model processors.

In *EXPRESS-I*, entity instances are represented in terms of attribute values: the traits or characteristics considered important for use and understanding. These attributes have a representation which might be a simple data type (such as integer) or another entity type. A geometric point might be defined in terms of three real numbers. Names are given to the attributes which contribute to the definition of an entity. Thus, for a geometric point the three real numbers might be named x, y and z. A relationship is established between the entity being defined and the attributes that define it, and in a similar manner between the attribute and its representation.

The *EXPRESS-I* instance language provides a means of displaying instantiations of *EXPRESS* data elements. The language is designed principally for human readability and for ease of generating *EXPRESS-I* element instances from definitions in an *EXPRESS* schema. Elsewhere in this International Standard, for example ISO 10303-21, there are specifications for computer efficient methods for instantiating a schema. *EXPRESS-I* is not intended to be a replacement for such methods.

The major elements of the language are shown in figure 1.

The language has two major parts. The first part is for the display of data instances. Data may be displayed on an entity by entity basis, on a schema basis or as a collection of schema instances which are taken to be a display of some model of a universe of discourse. Within the *EXPRESS-I* language these are called *object instances*, *schema instances* and a *model*. In figure 1 the model is assumed to have been defined using *EXPRESS*.

The second part of the language is for the specification of Abstract Test Cases for the purposes of formally describing tests to be performed against an implementation of an *EXPRESS* defined information model. The language constructs provided for this purpose are the *test case* and the *context*. This portion of the language also utilises the procedural aspects of the *EXPRESS* language. Instances of data may be parameterised and stored in a context. Many different test cases may assign values for the parameterised data in a context and use that data as part of their test specification.

The data instances resulting from the application of a test case may be displayed via the model portion of the language.

NOTE – The examples of *EXPRESS-I* usage in this manual do not conform to any particular style rules. Indeed, the examples sometimes use poor style to conserve space or to show flexibility. The examples are not intended to reflect the content of the information models defined in other parts of this International Standard. They are crafted to show particular features of *EXPRESS-I*. Any similarity between the examples and the normative information models or test cases specified in other parts of ISO 10303 should be ignored.

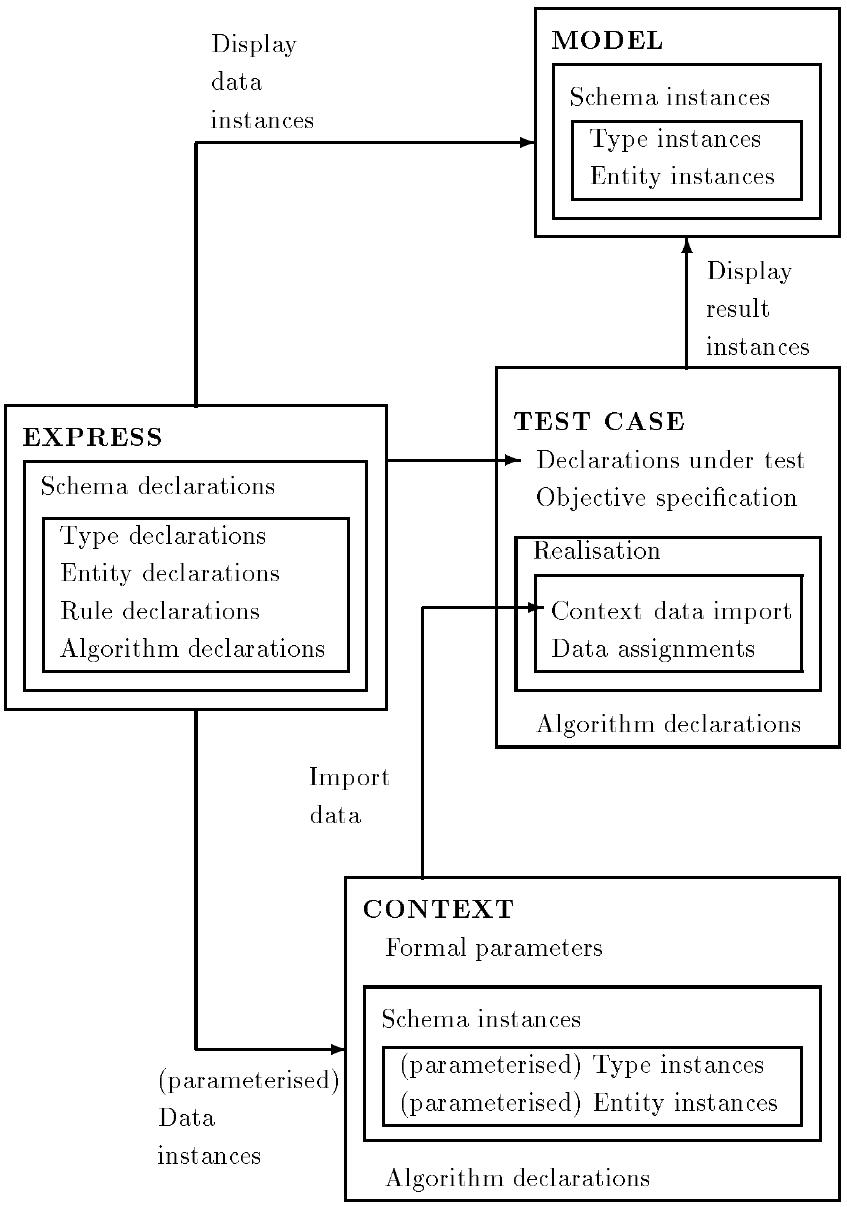

**Figure 1 – The major elements of the *EXPRESS-I* language.**

# Industrial automation systems and integration — Product data representation and exchange — Part 12 : Description methods: The EXPRESS-I language reference manual

## 1 Scope

This part of ISO 10303 defines a language by which an instance of (part of) a universe of discourse can be displayed. It also provides a formal description method for supporting the specification of abstract test cases. The language is called *EXPRESS-I*. It is a companion language to *EXPRESS* which is specified in ISO 10303-11.

*EXPRESS-I* is a an instantiation language for a conceptual schema language as defined in ISO TR 9007 and the particular conceptual schema language that formed the starting point for *EXPRESS-I* was *EXPRESS*. That is, it provides for the display of the state of the objects belonging to a universe of discourse and the information units pertaining to those objects.

The following are within the scope:

– display of instances of schemas;

– display of instances of types and entities;

– test case data;

– mapping from *EXPRESS* schemas and data types to *EXPRESS-I* instances.

The following are outside the scope of this part of ISO 10303:

– mapping from other (conceptual schema) languages to *EXPRESS-I*;

– definition of database formats;

– definition of file formats;

– definition of transfer formats;

– process control;

– information processing;

– exception handling.

*EXPRESS-I* is not a programming language.

# 2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 8824-1:—[1], *Information technology — Open systems interconnection — Abstract syntax notation one (ASN.1) — Part 1: Specification of basic notation.*

ISO 10303-1:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles.*

ISO 10303-11:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual.*

ISO/IEC 10646-1:1993, *Information technology — Universal multiple-octet coded character set (UCS) — Part 1: Architecture and basic multilingual plane.*

# 3 Definitions

## 3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1:

- Data;
- Information model.

## 3.2 Terms defined in ISO 10303-11

This part of ISO 10303 makes use of the following terms defined in ISO 10303-11:

- Complex entity data type;
- Complex entity instance;
- Constant;
- Data type;

[1] To be published.

- Entity;
- Entity instance;
- Instance;
- Object;
- Population;
- Property;
- Simple entity instance;
- Subtype/supertype graph;
- Token.

## 3.3 Other definitions

For the purposes of this part of ISO 10303, the following definitions apply:

**3.3.1 Attribute:** A trait, quality or property that is a characteristic of an entity.

**3.3.2 Conceptual schema:** A schema that is not configured for a specific implementation.

**3.3.3 Information:** Knowledge of facts, processes or ideas.

**3.3.4 Information base:** A collection of type instances, consistent with each other and with a conceptual schema, that hold for an instance of a universe of discourse.

**3.3.5 Model:** A formal description of a universe of discourse.

**3.3.6 Object Base:** An information base that is computer processible.

**3.3.7 Schema:** A collection of items forming part or all of a model.

**3.3.8 Type:** A representation of a domain of valid values.

**3.3.9 Universe of discourse:** All those real world objects that are of potential interest. These are a subset of all the real world objects.

# 4 Conformance requirements

## 4.1 Formal specifications written in EXPRESS-I

A formal specification written in *EXPRESS-I* shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level and all lower levels are verified for the specification.

### 4.1.1 Levels of checking

**Level 1:** Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule given in annex A. A formal specification is referentially valid if all references to *EXPRESS-I* items are consistent with the scope and visibility rules defined in clause 11.

**Level 2:** Type checking. This level consists of checking the formal specification to ensure that type compatability in expressions and assignments, as defined for level 2 checking in ISO 10303-11, are valid.

**Level 3:** Value checking. This level consists of checking the formal specification to ensure that it compies with level 3 checking defined in ISO 10303-11.

**Level 4:** Complete checking. This level consists of checking a formal specification to ensure that it complies with all statements of requirements as specified in this part of ISO 10303.

## 4.2 Implementations of EXPRESS-I

An implementation of an *EXPRESS-I* language parser shall be able to parse any formal specification written in *EXPRESS-I*, consistent with the constraints as specified in annex B associated with that implementation. An *EXPRESS-I* language parser shall be said to conform to a particular checking level (as defined in 4.1.1) if it can apply all checks required by the level (and any level below that) to a formal specification written in *EXPRESS-I*.

The implementor of an *EXPRESS-I* language parser shall state any constraints which the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented in the form specified by annex B for the purposes of conformance testing.

# 5 Fundamental Principles

It is assumed that the reader of this document is familiar with the *EXPRESS* language as specified in ISO 10303-11.

It is assumed that when *EXPRESS-I* is used to display entity instances that there is elsewhere a related set of entity definitions. It is further assumed that these will typically be described using the *EXPRESS* language.

# 6 Language elements

This clause specifies the basic elements from which sentences in the *EXPRESS-I* language are composed: the character set, remarks, symbols, reserved words, identifiers, literals and values.

The boxed syntax definitions in the body of this document are excerpts from the *EXPRESS-I* language syntax in annex A which defines the complete syntax of the language and provides any language productions not given here. The method of specifying the syntax is a superset of that used for *EXPRESS* as defined in clause 6 of ISO 10303-11.

NOTE 1 – For convenience of the reader, the *EXPRESS* method is repeated in annex D, together with the extensions for *EXPRESS-I*.

The basic language elements are composed into a stream of source text, typically broken into physical lines. A physical line is any number (including zero) of characters ended by a newline (see 6.1.5.2).

NOTE 2 – *EXPRESS-I* source is easier to read when statements are broken into lines and whitespace is used to set off different constructs.

## 6.1 Character set

*EXPRESS-I* source shall only use the characters defined by the following selected subset of ISO 10646; cells 00 to 7F of row 00 of plane 00 of group 00. This selected subset of ISO 10646 is called the *EXPRESS-I* character set. Members of this set are referred to by the cell of ISO 10646 in which these characters are defined, these cell numbers are specified in hexadecimal. The printable characters from this subset (cells 21–7E) are combined to form the tokens for the *EXPRESS-I* language. The *EXPRESS-I* tokens are keywords, identifiers, symbols, literals or values. The *EXPRESS-I* character set is further classified below:

NOTES

1 – The *EXPRESS-I* character set is the same as the *EXPRESS* character set.

2 – This clause only refers to the characters used to specify *EXPRESS-I* source, and does not specify the domain of characters allowed within a string value.

### 6.1.1 Digits

*EXPRESS-I* uses the Arabic digits 0–9 (cells 30–39 of the *EXPRESS-I* character set).

**Syntax:**

```
120 digit = < as EXPRESS > .
```

### 6.1.2 Letters

*EXPRESS-I* uses the upper and lower case letters of the English alphabet (cells 41–5A and 61–7A of the *EXPRESS-I* character set). The case of letters is significant only within explicit string values.

NOTE – *EXPRESS-I* may be written using upper, lower or mixed case letters.

**Syntax:**

```
124 letter = < as EXPRESS > .
```

### 6.1.3 Special characters

The special characters (printable characters which are neither letters nor digits) are used mainly for punctuation and as operators. Some of the special characters shown are not used as part of the language. They may be used within remarks and string values, however. These special characters are in cells 21–2F, 3A–3F, 40, 5B–5E, 60 and 7B–7E of the *EXPRESS-I* character set.

**Syntax:**

```
134 special = < as EXPRESS > .
```

### 6.1.4 Underscore

The underscore character (_, cell 5F of the *EXPRESS-I* character set) may be used in identifiers and keywords, with the exception that the underscore character shall not be used as the first character.

### 6.1.5 Whitespace

Whitespace is defined by the following sub-clauses and by 6.1.6. Whitespace shall be used to separate the tokens in *EXPRESS-I* source.

NOTE – Liberal, and consistent, use of whitespace can improve the structure and readability of *EXPRESS-I* source.

#### 6.1.5.1 Space character

One or more spaces (cell 20 of the *EXPRESS-I* character set) can appear between two tokens or within a string value. The notation `\s` is used to represent the space character in the syntax of the language.

#### 6.1.5.2 Newline

A newline marks the physical end of a line within a formal specification written in *EXPRESS-I*. Newline is normally treated as a space but is significant when it terminates a tail remark or appears within a string value. A newline is represented by the notation `\n` in the syntax of the language.

The representation of a newline is implementation defined.

#### 6.1.5.3 Other characters

Characters not defined in clause 6.1.1 to clause 6.1.5.2 (i.e., cells 00–1F and 7F of the *EXPRESS-I* character set) shall be treated as whitespace, unless within a string value. The notation `\o` is used to represent any of these other characters in the syntax of the language.

### 6.1.6 Remarks

A remark is used for documentation and shall be interpreted by an *EXPRESS-I* parser as whitespace. There are two forms of remark: embedded remark and tail remark.

#### 6.1.6.1 Embedded remark

The character pair `(*` denotes the start of an embedded remark and the character pair `*)` denotes its end. An embedded remark may appear between any two tokens.

**Syntax:**

```
142 embedded_remark = < as EXPRESS > .
```

Any character within the *EXPRESS-I* character set may occur between the start and end of an embedded remark, including the newline character; therefore embedded remarks can span several physical lines.

Embedded remarks may be nested.

NOTE – Care must be taken when nesting remarks to ensure that there are matched pairs of symbols.

EXAMPLE 1 – The following is an example of embedded nested remarks.

```
(* The '(*' symbol starts an embedded remark, and the '*)' symbol ends it. *)
```

#### 6.1.6.2 Tail remark

The tail remark is written at the end of a physical line. Two consecutive hyphens (`--`) start the tail remark and the following newline terminates it.

**Syntax:**

```
144 tail_remark = < as EXPRESS > .
```

EXAMPLE 2 – A tail remark

```
-- This is a tail remark and is ended by a newline
```

## 6.2 Reserved words

The reserved words of *EXPRESS-I* are the keywords and the names of built-in constants, functions and procedures. The reserved words shall not be used as identifiers. The reserved words of *EX-*

*PRESS-I* are described below.

### 6.2.1 Keywords

*EXPRESS-I* uses a subset of the *EXPRESS* keywords, together with some additional ones.

Table 1 lists the keywords that are common to both *EXPRESS-I* and *EXPRESS*. Table 2 lists the additional *EXPRESS-I* keywords.

NOTE – Keywords have an uppercase production which represents the literal. This is to enable easier reading of the syntax productions.

**Table 1 – Keywords common to *EXPRESS-I* and *EXPRESS*.**

| | | | |
|---|---|---|---|
| ABSTRACT | AGGREGATE | ALIAS | ARRAY |
| BAG | BEGIN | BINARY | BOOLEAN |
| BY | CASE | CONSTANT | CONTEXT |
| DERIVE | ELSE | END | END_ALIAS |
| END_CASE | END_CONSTANT | END_CONTEXT | END_ENTITY |
| END_FUNCTION | END_IF | END_LOCAL | END_MODEL |
| END_PROCEDURE | END_REPEAT | END_TYPE | ENTITY |
| ENUMERATION | ESCAPE | FIXED | FOR |
| FUNCTION | GENERIC | IF | INTEGER |
| INVERSE | LIST | LOCAL | LOGICAL |
| MODEL | NUMBER | OF | ONEOF |
| OPTIONAL | OTHERWISE | PROCEDURE | QUERY |
| REAL | REPEAT | RETURN | SELECT |
| SET | SKIP | STRING | SUBTYPE |
| SUPERTYPE | THEN | TO | TYPE |
| UNIQUE | UNTIL | VAR | WHERE |
| WHILE | | | |

**Table 2 – Additional *EXPRESS-I* keywords**

| | | | |
|---|---|---|---|
| CALL | CRITERIA | END_CALL | END_CRITERIA |
| END_NOTES | END_OBJECTIVE | END_PARAMETER | END_PURPOSE |
| END_REALIZATION | END_REFERENCES | END_SCHEMA_DATA | END_TEST_CASE |
| IMPORT | NOTES | OBJECTIVE | PARAMETER |
| PURPOSE | REALIZATION | REFERENCES | SCHEMA_DATA |
| SUBOF | SUPOF | TEST_CASE | USING |
| WITH | | | |

### 6.2.2 Reserved words which are operators

The operators defined by reserved words are shown in table 3. These are the same as the *EXPRESS* operators and are defined in clause 12 of ISO 10303-11.

**Table 3 – The *EXPRESS-I* use of *EXPRESS* operators.**

| | | | |
|---|---|---|---|
| AND | ANDOR | DIV | IN |
| LIKE | MOD | NOT | OR |
| XOR | | | |

### 6.2.3 Built-in constants

**Syntax:**

```
48i Constant = LogicalValue | MathConstant | Nil .
84i LogicalValue = logical_literal .
242 logical_literal = < as EXPRESS > .
84i MathConstant = CONST_E | PI .
30i Nil = '?' .
```

The names of the *EXPRESS-I* built-in constants are given in table 4. These are the same as the *EXPRESS* constants and are defined in clause 14 of ISO 10303-11.

**Table 4 – The *EXPRESS-I* use of *EXPRESS* constants.**

| | | | |
|---|---|---|---|
| ? | CONST_E | FALSE | PI |
| SELF | TRUE | UNKNOWN | |

The question mark character (?) represents the notion of a Nil, or unspecified, value.

### 6.2.4 Built-in functions

The names of the *EXPRESS* functions that may be used within *EXPRESS-I* are given in table 5.

**Table 5 – The *EXPRESS-I* use of *EXPRESS* functions.**

| | | | |
|---|---|---|---|
| ABS | ACOS | ASIN | ATAN |
| BLENGTH | COS | EXISTS | EXP |
| FORMAT | HIBOUND | HIINDEX | LENGTH |
| LOBOUND | LOG | LOG10 | LOG2 |
| LOINDEX | NVL | ODD | ROLESOF |
| SIN | SIZEOF | SQRT | TAN |
| TYPEOF | USEDIN | VALUE | VALUE_IN |
| VALUE_UNIQUE | | | |

The definitions of these functions are given in clause 15 of ISO 10303-11.

### 6.2.5 Built-in procedures

The names of the *EXPRESS* procedures that may be used within *EXPRESS-I* are given in table 6. The procedures are defined in clause 16 of ISO 10303-11.

**Table 6 – The *EXPRESS-I* use of *EXPRESS* procedures.**

| | |
|---|---|
| INSERT | REMOVE |

## 6.3 Symbols

Symbols are special characters or groups of special characters which have a special meaning in *EXPRESS-I*. Symbols are used in *EXPRESS-I* as delimeters and operators. A delimeter is used to begin, seperate or terminate adjacent lexical or syntactic elements. Interpretation of these elements would be impossible without separators. Operators denote that actions shall be performed on the operands which are associated with the operator. The *EXPRESS-I* symbols are shown in table 7 and table 8.

**Table 7 – Symbols common to *EXPRESS-I* and *EXPRESS*.**

| | | | |
|---|---|---|---|
| `.` | `,` | `;` | `:` |
| `*` | `+` | `-` | `=` |
| `%` | `'` | `\` | `/` |
| `<` | `>` | `[` | `]` |
| `{` | `}` | `\|` | `e` |
| `(` | `)` | `<=` | `<>` |
| `>=` | `<*` | `:=` | `\|\|` |
| `**` | `--` | `(*` | `*)` |
| `:=:` | `:<>:` | | |

**Table 8 – Additional *EXPRESS-I* symbols.**

| | | | |
|---|---|---|---|
| `@` | `!` | `->` | `<-` |
| `==` | `"` | | |

## 6.4 Identifiers and references

Identifiers are names given to the elements declared in an *EXPRESS-I* instantiation. An identifier shall not be the same as an *EXPRESS-I* or *EXPRESS* reserved word.

**Syntax:**

```
187  constant_id = < as EXPRESS > .
146  constant_ref = < as EXPRESS > .
198  entity_id = < as EXPRESS > .
282  schema_id = < as EXPRESS > .
140  simple_id = < as EXPRESS > .
154  type_ref = < as EXPRESS > .
54i  ContextId = simple_id .
35i  ContextRef = ContextId .
64i  EntityInstanceId = simple_id .
36i  EntityInstanceRef = '@' EntityInstanceId .
68i  EnumerationId = type_ref .
64i  EnumerationInstanceId = simple_id .
37i  EnumerationInstanceRef = '@' EnumerationInstanceId .
87i  ModelId = simple_id .
91i  ObjectInstanceRef = EntityInstanceRef | EnumerationInstanceRef |
                        SelectInstanceRef | SimpleInstanceRef |
                        TypeInstanceRef .
95i  ParameterId = simple_id .
38i  ParameterRef = ParameterId .
110i SelectId = type_ref .
112i SelectInstanceId = simple_id .
39i  SelectInstanceRef = '@' SelectInstanceId .
116i SimpleInstanceId = simple_id .
40i  SimpleInstanceRef = '@' SimpleInstanceId .
122i TestCaseId = simple_id .
129i TypeId = type_ref .
131i TypeInstanceId = simple_id .
41i  TypeInstanceRef = '@' TypeInstanceId .
```

The first character of a simple identifier shall be a letter. The remaining characters, if any, may be any combination of letters, digits and the underscore character. Identifiers shall not have any embedded white space.

The implementor of an *EXPRESS-I* language parser shall specify the maximum number of characters of an identifier which can be read by that implementation, using annex B.

NOTE – The letters used to form identifiers are not case sensitive as upper and lower case letters are treated as equal.

EXAMPLE 3 – Valid simple identifiers

```
POINT  line   Circle  AnEntity  item567  An_integer
```

EXAMPLE 4 – Invalid simple identifiers

```
_POINT       underscore can't be first character
line?        ? can't be part of identifier
3dThing      digit can't be first character
Pi           Pi is an EXPRESS-I keyword
```

An element may be referenced via its identifier. Constant, parameter, and model elements are referenced via the corresponding identifier.

The first character of an entity, enumeration, type or select instance reference shall be `@` followed by at least one character. The characters after the intial `@` can be any combination of letters, digits, and the underscore character which form a valid entity, enumeration, select or type instance identifier. Collectively, these are termed object instance references.

EXAMPLE 5 – Valid object instance references

```
@POINT   @line     @Circle  @AnEntity  @item567
```

EXAMPLE 6 – Invalid object instance references

```
@line?        ? can't be part of identifier
3dThing       @ must be first character
@subof        subof is an EXPRESS-I keyword
@@Circle      @ must appear only as the first character
@567          characters following the @ must begin with a letter
```

# 7 Named domains

This clause defines the domain types provided as part of the language. Domains are used to delineate the allowable instance values. A named domain is an entity, a type, an enumeration, or a select domain.

## 7.1 Entity domain

An entity domain represents a class of objects which have common attributes.

**Syntax:**

```
61i  EntityDomain = [ SchemaId '.' ] EntityId .
```

NOTE – An entity domain corresponds to an *EXPRESS* ENTITY data type.

## 7.2 Enumeration domain

An enumeration domain has as its domain an ordered set of names.

**Syntax:**

```
67i  EnumerationDomain = [ SchemaId '.' ] EnumerationId .
```

NOTE – An enumeration domain corresponds to an *EXPRESS* ENUMERATION data type.

## 7.3 Select domain

A select domain has as its domain a union of domains.

**Syntax:**

```
109i SelectDomain = [ SchemaId '.' ] SelectId .
```

NOTE – A select domain corresponds to an *EXPRESS* SELECT data type.

## 7.4 Type domain

A type domain is an extension to the other domains in the language.

**Syntax:**

```
128i TypeDomain = [ SchemaId '.' ] TypeId .
```

NOTE – A type domain corresponds to an *EXPRESS* defined data TYPE which is neither an ENUMERATION nor a SELECT.

## 8 Values and instances

The clause describes the *EXPRESS-I* instantiation capabilities.

## 8.1 Base values

**Syntax:**

```
45i  BaseValue = SimpleValue | EnumerationValue .
117i SimpleValue = BinaryValue | BooleanValue | LogicalValue |
                   NumberValue | StringValue .
```

A simple value is a self defining constant value. The domain of the value depends on how characters are composed to form a token.

### 8.1.1 Binary value

A binary value represents the value of a binary domain.

**Syntax:**

```
25i  BinaryValue = binary_literal .
136  binary_literal = < as EXPRESS > .
```

A binary value is composed of the `%` character followed by one or more bits (0 or 1).

The implementor of an *EXPRESS-I* language parser shall specify the maximum number of bits in a binary value which can be read by that implementation, using annex B.

EXAMPLE 7 – A valid binary value

```
%10100110000101
```

### 8.1.2 Boolean value

A boolean value represents the value of a boolean domain.

**Syntax:**

```
47i  BooleanValue = TRUE | FALSE .
```

A boolean value is one of the built-in constants FALSE or TRUE.

### 8.1.3 Number value

A number value is either an integer value or a real value.

**Syntax:**

```
89i  NumberValue = IntegerValue | RealValue .
```

### 8.1.4 Integer value

An integer value represents the value of an integer domain.

**Syntax:**

```
29i IntegerValue = [ sign ] integer_literal .
138 integer_literal = < as EXPRESS > .
286 sign = < as EXPRESS > .
```

An integer literal is composed entirely of digits. An integer value is composed of an integer literal, optionally preceeded by a sign. It defines a positive, negative or zero integer (whole) number.

The implementor of an *EXPRESS-I* language parser shall specify the maximum value of an integer value which can be read by that implementation, using annex B.

EXAMPLE 8 – Valid integer values

```
0  1  -1  891562934527619
```

EXAMPLE 9 – Invalid integer values

```
1.0              can't include a decimal point
```

### 8.1.5 Logical value

A logical value represents the value of a logical domain.

**Syntax:**

```
83i LogicalValue = logical_literal .
242 logical_literal = < as EXPRESS > .
```

A logical value is one of the built-in constants FALSE, TRUE or UNKNOWN.

## 8.1.6 Real value

A real value represents the value of a real domain.

A real value is either a signed math constant or a signed real literal.

**Syntax:**

```
99i RealValue =  SignedMathConstant | SignedRealLiteral .
31i SignedMathConstant = [ sign ] MathConstant .
84i MathConstant = CONST_E | PI .
32i SignedRealLiteral = [ sign ] real_literal .
139 real_literal = < as EXPRESS > .
```

A signed math constant is one of the built-in mathematical constants (i.e $e$ or $\pi$) optionally preceeded by a sign.

The mathematical constant $e = 2.7182\ldots$ is represented by the *EXPRESS-I* constant CONST_E.

The mathematical constant $\pi = 3.1415\ldots$ is represented by the *EXPRESS-I* constant PI.

EXAMPLE 10 – Signed math constants

```
-const_e      Pi
```

A signed real literal is composed of a (signed) mantissa and an optional exponent. It defines a rational number.

The implementor of an *EXPRESS-I* language parser shall specify the maximum precision and maximum exponent of a real value which can be read by that implementation, using annex B.

EXAMPLE 11 – Valid real values

```
0.0  -1.E6  1.e-6  8915629.34527619
```

EXAMPLE 12 – Invalid real values

```
.001              must have at least one digit before the point
1e10              must have a decimal point in the mantissa
1.0e-12.0         can't have a decimal point in the exponent
CONSTE            mispelled built in constant
```

## 8.1.7 String value

A string value represents the value of a string domain. There are two forms of string value, the explicit string value and encoded string value. An explicit string value is composed of a sequence of characters in the *EXPRESS-I* character set enclosed by quote marks ('). A quote mark within an explicit string value is represented by two consecutive quote marks. An encoded string value is a four octet encoded representation of a sequence of characters in ISO 10646 enclosed in double quote marks ("). The encoding is defined as follows:

- first octet = ISO 10646 group in which the character is defined;
- second octect = ISO 10646 plane in which the character is defined;
- third octect = ISO 10646 row in which the character is defined;
- fourth octect = ISO 10646 cell in which the character is defined.

**Syntax:**

```
118i StringValue = SimpleStringValue | EncodedStringValue .
33i  SimpleStringValue = \q { ( \q \q ) | not_quote | \s | \o | \n } \q .
130  not_quote = < as EXPRESS > .
27i  EncodedStringValue = '"' { encoded_character | \n } '"' .
122  encoded_character = < as EXPRESS > .
```

The implementor of an *EXPRESS-I* language parser shall specify the maximum number of characters of a string value which can be read by that implementation, using annex B.

The implementor of an *EXPRESS-I* language parser shall also specify the maximum number of octets (must be a multiple of four) of an encoded string value which can be read by that implementation, using annex B.

NOTE – An *EXPRESS-I* string value differs from an *EXPRESS* string literal, as in the former case a string value may span more than one physical line, whereas an *EXPRESS* string literal cannot span more than one physical line.

EXAMPLE 13 – Valid explicit string values

```
'This is a string on one line.'
'This
      is
         a
           multiline
                     string.'

'This string''s got a single quote mark embedded in it.'
```

Reads . . . This string's got a single quote mark embedded in it.

EXAMPLE 14 – Invalid explicit string values

```
'This string is invalid because there is no closing quote mark.
```

EXAMPLE 15 – Valid encoded string values

```
"00000041"
```

Reads . . . A.

```
"000000C5"
```

Reads . . . Å

EXAMPLE 16 – Invalid encoded string values

```
"000041"
```

Octets must be supplied in groups of four

```
"00000041 000000C5"
```

Can't have a space between octets

### 8.1.8 Enumeration value

An enumeration value represents a value of an enumeration domain.

**Syntax:**

```
28i  EnumerationValue = '!' simple_id .
```

An enumeration value is a simple identifier prepended with an exclamation mark (`!`). A simple identifier is a character sequence of letters, digits and underscore, with the first character being a letter.

EXAMPLE 17 – Valid enumeration values

```
!red  !green  !forward
```

## 8.2 Aggregation values

*EXPRESS-I* distinguishes two forms of aggregation of values — fixed and dynamic. A fixed aggregation is an aggregation of like things, where the number of items in the aggregation is constant. A dynamic aggregation is an aggregation of like things, where the number of items in the aggregation may be variable. Aggregation values may be nested.

**Syntax:**

```
43i  AggregationValue = DynamicAggr | FixedAggr .
57i  DynamicAggr = '(' [ DynamicList ] ')' .
59i  DynamicList = DynamicMember { ',' DynamicMember } .
60i  DynamicMember = AggregationValue | ConstantValue |
                     DerattValue | ParmValue | ReqattValue |
                     TypeValue .
74i  FixedAggr = '[' FixedList ']' .
75i  FixedList = FixedMember { ',' FixedMember } .
76i  FixedMember = DynamicMember | Nil .
```

The allowable domains of the elements within the aggregation depend on the domain context. These contexts are:

- Constants (see clause 8.9);
- Derived attributes (see clause 8.7.1.2);
- Explicit attributes (see clause 8.7.1.1);
- Parameters (see clause 9.2.2);
- Defined data types (see clause 8.4).

**Rules and restrictions:**

a) Elements within a dynamic aggregation shall not be `Nil`.

b) Elements within a fixed aggregation may be `Nil`.

c) The element values within an aggregation shall be compatible with the aggregation domain.

EXAMPLE 18 – Aggregation values

```
(10,-10,0)       a dynamic aggregation of 3 integer values
(1,1,2,2,3,3)    a dynamic aggregation of 6 integer values
()               an empty dynamic aggregation
[1,2,3,4]        a fixed aggregation of 4 integer values
([1,2],[3,?])    a dynamic aggregation of a fixed aggregation of 2 values
```

## 8.3 Simple instance

A simple instance is a representation of the value of one instance of a simple value.

**Syntax:**

```
115i SimpleInstance = SimpleInstanceId '=' SimpleValue ';' .
116i SimpleInstanceId = simple_id .
117i SimpleValue = BinaryValue | BooleanValue | LogicalValue |
                   NumberValue | StringValue .
40i  SimpleInstanceRef = '@' SimpleInstanceId .
```

EXAMPLE 19 – Some simple instances

```
r1 = 27.0;
s1 = 'A string';
```

## 8.4 Type instance

A type instance is a representation of the value of one instance of a TYPE domain.

**Syntax:**

```
130i TypeInstance = TypeInstanceId '=' TypeInstanceValue ';' .
131i TypeInstanceId = simple_id .
132i TypeInstanceValue = TypeDomain '{' TypeValue '}' .
133i TypeValue = AggregationValue | BaseValue | ConstantRef |
                 EntityInstanceValue | NamedInstanceValue |
                 ObjectInstanceRef | ParameterRef .
41i  TypeInstanceRef = '@' TypeInstanceId .
```

**Rules and restrictions:**

a) The value of the instance shall be either a simple value, an entity instance reference, a type instance reference, or aggregations of these.

EXAMPLE 20 – Some type instances

```
t1 = a_real{27.0};
t2 = an_array_of_string{['one', 'two']};
t3 = a_dynamic_aggregate_of_integer{(1,1,2,3,5,8,13)};
```

## 8.5 Select instance

A select instance is a representation of the value of one instance of a SELECT domain.

**Syntax:**

```
111i SelectInstance = SelectInstanceId '=' SelectInstanceValue ';' .
112i SelectInstanceId = simple_id .
113i SelectInstanceValue = SelectDomain '{' SelectValue '}' .
114i SelectValue = EnumerationValue | NamedInstanceValue |
                   ObjectInstanceRef | TypeValue .
39i  SelectInstanceRef = '@' SelectInstanceId .
```

**Rules and restrictions:**

a) The value of the instance shall be either a type instance reference, a select instance reference, an enumeration instance reference, or an entity instance reference.

EXAMPLE 21 – A select instance

```
s1 = type_or_entity{@e27};
```

## 8.6 Enumeration instance

An enumeration instance is a representation of the value of one instance of an ENUMERATION domain.

**Syntax:**

```
69i  EnumerationInstance = EnumerationInstanceId '='
                           EnumerationInstanceValue ';' .
70i  EnumerationInstanceId = simple_id .
71i  EnumerationInstanceValue = EnumerationDomain
                                '{' EnumerationValue '}' .
28i  EnumerationValue = '!' simple_id .
37i  EnumerationInstanceRef = '@' EnumerationInstanceId .
```

**Rules and restrictions:**

a) The value of the instance shall be an enumeration value.

EXAMPLE 22 – Some enumeration instances

```
enum1 = an_enum{!first};
enum2 = an_enum{!second};
```

## 8.7 Entity instance

An entity instance is a representation of one instantiation of an ENTITY domain.

**Syntax:**

```
63i  EntityInstance = EntityInstanceId '=' EntityInstanceValue ';' .
64i  EntityInstanceId = simple_id .
65i  EntityInstanceValue = EntityDomain '{' [ InheritsFrom ]
                           { ExplicitAttr } { DerivedAttr }
                           { InverseAttr } [ BequeathsTo ] '}' .
36i  EntityInstanceRef = '@' EntityInstanceId .
```

### 8.7.1 Attributes

An *EXPRESS-I* entity instance may have zero or more attributes. Attributes are classified into explicit, derived and inverse attributes.

EXAMPLE 23 – Empty entity instances

```
e2 = ent_inst{};
eg = ent_inst{};
```

#### 8.7.1.1 Explicit attributes

An explicit attribute is a required property of an entity.

**Syntax:**

```
72i  ExplicitAttr = RequiredAttr | OptionalAttr .
101i RequiredAttr = RoleName '->' ( ReqattValue | Nil ) ';' .
94i  OptionalAttr = RoleName '->' OptattValue ';' .
102i RoleName = attribute_ref .
100i ReqattValue = AggregationValue | BaseValue | ConstantRef |
                   NamedInstanceValue | ObjectInstanceRef |
                   ParameterRef | SelectValue | TypeValue .
91i  ObjectInstanceRef = EntityInstanceRef | Enumeration InstanceRef |
                         SelectInstanceRef | TypeInstanceRef |
                         SimpleInstanceRef .
88i  NamedInstanceValue = EnumerationInstanceValue |
                          SelectInstanceValue | TypeInstanceValue .
93i  OptattValue = ReqattValue | Nil .
30i  Nil = '?' .
```

An explicit attribute consists of the attribute role name, followed by the symbol `->`, followed by the value of the domain of the role, and finally completed by a semi-colon. The value of the role domain for a required attribute may be a reference to an entity or type instance, a value, a named value, a constant or a parameter, or aggregates of these. The value of the role domain for an optional attribute is the same as for a required attribute, with additionaly a `Nil` value for when the value is not defined.

NOTE – An explicit attribute may be given a `Nil` value. In this case, if the entity definition is based upon an *EXPRESS* ENTITY then the instance is not conforming to the *EXPRESS* definition.

EXAMPLE 24 – Explicit attributes

```
a_real         -> 1.2;
an_integer     -> 3;
a_list         -> (1,2,3);
a_boolean      -> TRUE;
a_logical      -> UNKNOWN;
an_enumeration -> !enum1;
a_string       -> 'A string';
entity_ref     -> @instance2;
optional_str   -> ?;
optional_int   -> 42;
a_parameter    -> par1;
a_constant     -> c1;
```

#### 8.7.1.2 Derived attribute

A derived attribute is one whose value can be calculated from the values of other properties of an entity.

**Syntax:**

```
56i  DerivedAttr = RoleName [ '<-' DerattValue ] ';' .
102i RoleName = attribute_ref .
55i  DerattValue = AggregationValue | BaseValue | EntityInstanceRef |
                   EntityInstanceValue | EnumerationInstanceValue |
                   TypeInstanceRef | TypeInstanceValue | TypeValue .
```

A derived attribute consists of the attribute role name, optionally followed by the symbol **<-** and the value of the domain of the role, and finally completed by a semi-colon. The value of the role domain may be a reference to an entity or type instance, a value, a constant, or aggregates of these. Alternately, the value may be `Nil` in the case where the value is not defined.

EXAMPLE 25 – Derived attributes

```
a_real         <- 1.2;
an_integer     <- 3;
a_boolean      <- TRUE;
a_logical;
an_enumeration <- !enum1;
a_string       <- 'A string';
entity_ref     <- @instance2;
null_derived   <- ?;
```

### 8.7.1.3 Inverse attribute

If an entity instance has established a relationship with the current entity instance via referencing the current instance in an explicit attribute, then an inverse attribute may be used to describe that relationship in the context of the current instance.

**Syntax:**

```
82i  InverseAttr = RoleName [ '<-' InvattValue ] ';' .
102i RoleName = attribute_ref .
81i  InvattValue = DynamicEntityRefList .
58i  DynamicEntityRefList = '(' [ EntityRefList ] ')' .
66i  EntityRefList = EntityInstanceRef { ',' EntityInstanceRef } .
36i  EntityInstanceRef = '@' EntityInstanceId .
```

An inverse attribute consists of the attribute role name, optionally followed by the symbol **<-** and the value of the domain of the role, and finally completed by a semi-colon. The value of the role domain is a (possibly empty) dynamic list of entity instance references.

EXAMPLE 26 – Inverse attributes

```
inverse_1 <- (@a1, @b3);
inverse_2;
inverse_3 <- ();
```

### 8.7.2 Supertypes and subtypes

An *EXPRESS-I* entity instance inherits attributes and their values from its SUPERTYPE instances (if any) and bequeathes attributes and their values to its SUBTYPE instances (if any).

**Syntax:**

```
46i  BequeathsTo = SUPOF DynamicEntityRefList  ';' .
80i  InheritsFrom = SUBOF DynamicEntityRefList ';' .
```

Supertype instances are referenced following the SUBOF keyword and are enclosed in parentheses.

Subtype instances are referenced following the SUPOF keyword and are enclosed in parentheses.

EXAMPLE 27 – Supertypes and subtypes

```
i1 = super{super_int -> 2; SUPOF(@s1); };
s1 = sub{SUBOF(@i1); sub_real -> 23.7; };

i2 = super{super_int -> 7; SUPOF(@s2); };
s2 = sub{SUBOF(@i2); sub_real -> -42.0; };
```

## 8.8 Schema data instance

A SCHEMA_DATA instance defines an instance of (part of) a representation of a universe of discourse in which the elements declared have a related meaning and purpose. For example, `geometry` might be the name of a schema that collects instances of points, curves, surfaces, and other related elements. The order in which instances are declared in a schema instance is arbitrary.

**Syntax:**

```
104i SchemaInstanceBlock = SCHEMA_DATA SchemaId ';'
                           [ SchemaInstanceBody ] END_SCHEMA_DATA ';' .
103i SchemaId = schema_ref .
105i SchemaInstanceBody = [ ConstantBlock ] { ObjectInstance } .
90i  ObjectInstance = EntityInstance | EnumerationInstance |
                      SelectInstance | TypeInstance | SimpleInstance .
```

A schema instance declaration creates a new scope in which the following elements may be declared:

– Constants;

– Entity instances;

– Enumeration instances;

– Select instances;

– Type instances.

EXAMPLE 28 – An instantiation of an *EXPRESS* defined schema.

```
SCHEMA_DATA whatsits;

  (* EXPRESS defined constants *)
  CONSTANT
    one == 1.0;
    twopi == 6.2831853;
  END_CONSTANT;

  (* EXPRESS defined types *)
  n1 = name{('Joe','E','Bloggs')};
  n2 = name{('Mary','Jones')};

  (* EXPRESS defined entities *)
  p1 = point{x -> one; y -> twopi;};
  s1 = affianced{him -> @n1; her -> @n2;};

END_SCHEMA_DATA;
```

## 8.9 Constant instance

A constant declaration may be used to declare named constants. The scope of the constant identifiers declared within a constant block shall be the schema in which the constant block occurs. A named constant appearing in a constant declaration has an explicit initialization; the value of a constant cannot be modified after initialisation. Any occurance of the named constant outside the constant declaration shall be equivalent to an occurance of the initial value itself.

**Syntax:**

```
48i  ConstantBlock = CONSTANT { ConstantSpec } END_CONSTANT ';' .
50i  ConstantSpec = ConstantId '==' ConstantValue ';' .
49i  ConstantId = constant_ref .
51i  ConstantValue = AggregationValue | BaseValue | EntityInstanceValue |
                     NamedInstanceValue | SelectValue | TypeValue .
34i  ConstantRef = ConstantId .
```

The value of a constant may be an aggregation of values.

**Rules and restrictions:**

a) Each value shall be a simple value, an entity instance value, an enumeration value, or aggregations of these.

b) A named constant may appear in the declared value of another named constant.

EXAMPLE 29 – A CONSTANT block

```
CONSTANT
  zero     == 0.0;
  thousand == 1000;
  origin   == point{x -> zero; y -> zero;};
  large_circle == circle{center -> origin; radius -> thousand;};
  z_axis   == [0.0, 0.0, 1.0];
END_CONSTANT;
```

## 8.10 Model display

A MODEL defines one particular instantiation of a representation of a universe of discourse in which the elements have related meaning and purpose.

**Syntax:**

```
85i  ModelBlock = MODEL ModelId ';' ModelBody END_MODEL ';' .
87i  ModelId = simple_id .
86i  ModelBody = { SchemaInstanceBlock } .
38i  ModelRef = ModelId .
```

An *EXPRESS-I* MODEL declaration creates a new scope in which the following elements may be declared:

– Schema instances.

The intended usage of a MODEL is to exhibit the population of an object base.

EXAMPLE 30 – For instance, `bugatti_35` might be the name of a MODEL that contains data representing a car of type *Bugatti Type 35*. There may be several schema instances within this MODEL; one, say, for the blueprints of the car, and another containing maintenance data on the car type.

**Rules and restrictions:**

a) Each schema data instance within a MODEL shall be an instance of a different SCHEMA.

b) Each instance identifier within a MODEL shall be unique.

c) Values within a model shall not be parameter references.

EXAMPLE 31 – A skeleton MODEL.

```
MODEL a_model;

  SCHEMA_DATA a_schema;
  ...
  END_SCHEMA_DATA;

  SCHEMA_DATA another_schema;
  ...
  END_SCHEMA_DATA;
END_MODEL;
```

# 9 Test case specification

This clause describes the principal *EXPRESS-I* language elements related to the specification of test cases.

## 9.1 Context

A CONTEXT defines data instances and algorithms relevant to a representation of a universe of discourse in which the elements have related meaning and purpose. The data instances may be parameterised.

**Syntax:**

```
52i  ContextBlock = CONTEXT ContextId ';' ContextBody END_CONTEXT ';' .
54i  ContextId = simple_id .
53i  ContextBody = { SchemaReferenceSpec } [ FormalParameterBlock ]
                  { SchemaInstanceBlock | SupportAlgorithm } .
35i  ContextRef = ContextId .
```

An *EXPRESS-I* CONTEXT declaration creates a new scope in which the following elements may be declared:

- References to *EXPRESS* schemas (see clause 10.2);
- Formal parameters;
- Schema data instances;
- *EXPRESS* functions;
- *EXPRESS* procedures.

EXAMPLE 32 – For instance, `bugatti` might be the name of a CONTEXT that contains parameterised (i.e generic) data representing a car of type *Bugatti*. There may be several schema instances within this CONTEXT; one, say, for the blueprints of the car, and another containing maintenance data on the car type.

**Rules and restrictions:**

a) Each schema data instance within a CONTEXT shall be an instance of a different SCHEMA.

b) Each identifier within a CONTEXT shall be unique.

EXAMPLE 33 – A skeleton CONTEXT.

```
CONTEXT parameterised_model;

  PARAMETER
  ...
  END_PARAMETER;

  SCHEMA_DATA a_schema;
  ...
```

```
    END_SCHEMA_DATA;

    SCHEMA_DATA another_schema;
    ...
    END_SCHEMA_DATA;
  END_CONTEXT;
```

## 9.2 Parameters

A context can have formal parameters. Each formal parameter has a name and a domain. The name is an identifier that shall be unique within the scope of the context.

A test case can have actual parameters that provide specific values for the relevant formal parameters within a context.

To allow a generalization of the data types used to pass values to contexts there are the domains AGGREGATE and GENERIC. Conformant arrays may also be used to allow the generalization of array domains.

### 9.2.1 Formal parameter

A formal parameter may have a default value, which shall be compatible with the domain. Formal parameters that do not have default values are initialised to `Nil`.

**Syntax:**

```
78i  FormalParameterBlock = PARAMETER
                            { FormalParameter } END_PARAMETER ';' .
77i  FormalParameter = ParameterId ':' parameter_type
                       [ ':=' ParmValueDefault ] ';' .
95i  ParameterId = simple_id .
253  parameter_type = < as EXPRESS > .
98i  ParmValueDefault = AggregationValue | BaseValue | ConstantRef |
                        EntityInstanceValue | NamedInstanceValue |
                        ObjectInstanceRef | SelectValue | TypeValue |
                        expression .
204  expression = < as EXPRESS > .
38i  ParameterRef = ParameterId .
```

As there may be more than one schema data instance in a context containing parameters, it may happen that two or more of these schemas have entities or types with the same name but differing semantics. The use of one of these names as the domain identifier for a parameter would then be ambiguous. In this case, the name is qualified by prepending the schema name to the id with a dot as a seperator.

EXAMPLE 34 – A PARAMETER block.

```
  PARAMETER
    iv1       : INTEGER := 1;
    bv1       : BOOLEAN;
    p1        : name := name{first --> 'John'; last --> 'Doe';
```

```
                             married --> bv1;};
    p2        : name := name('Mary','Smith',TRUE);
    a_list    : LIST OF REAL := (0.0, 1.0, 2.0);
    a_set     : SET OF STRING;
    a_select  : selection := wheeled_vehicle;
    from_sch1 : sch1.vector := [1.0,3,0];
    from_sch2 : sch2.vector := [3.0,4.0,-0.5];
  END_PARAMETER;
```

### 9.2.2 Actual parameter

An actual parameter consists of a reference to a formal parameter, and a value for the parameter. The value shall be compatible with the domain of the formal parameter. The value overrides the default parameter value associated with the formal paramater.

**Syntax:**

```
42i  ActualParameter = ParameterRef ':=' ParmValue .
38i  ParameterRef = ParameterId .
97i  ParmValue = ObjectInstanceRef | expression .
204  expression = < as EXPRESS > .
```

EXAMPLE 35 – This shows some actual parameters for the formal parameters given in example 34.

```
    iv1       := 771;
    bv1       := FALSE;
    p1        := name('John', 'Smith', bv1);
    a_list    := [20.0, 1.0, 20.0, 33.72];
    a_set     := ['alpha', 'to', 'omega'];
    a_select  := @v23;
    from_sch1 := [0.0, -1.0];
    from_sch2 := [0.5, -0.2, -0.15];
```

## 9.3 Test case

A TEST_CASE specifies both administrative and instance data which may be used for the purposes of an abstract test case.

**Syntax:**

```
120i TestCaseBlock = TEST_CASE TestCaseId ';'
                     TestCaseBody END_TEST_CASE ';' .
122i TestCaseId = simple_id .
121i TestCaseBody = SchemaReferences ObjectiveBlock TestRealization
                   { SupportAlgorithm } .
106i SchemaReferences = SchemaReferenceSpec { SchemaReferenceSpec } .
```

A TEST_CASE declaration creates a new scope in which the following items may be declared or referenced:

– The items under test (see clause 10.2);

- The test objective;
- The test realization;
- Supporting algorithms.

A TEST_CASE references one or more *EXPRESS* SCHEMAs. It may reference a set of CONTEXTs, and possibly a set of parameter values, for the purposes of defining a set of test data.

**Rules and restrictions:**

a) The value of each actual parameter declared in a test case shall be compatible with the domain of the corresponding formal parameter declared in the context.

b) The test case value associated with each formal parameter in the context shall be that declared as the actual parameter, or the default value of the formal parameter if an actual parameter is not declared.

c) Data types within a test case shall be restricted to those type definitions specified within the referenced schemas.

## 9.4 Test objective

An OBJECTIVE is administrative data which may be used for an abstract test case.

**Syntax:**

```
92i  ObjectiveBlock = OBJECTIVE { TestPurpose } { TestReference }
                      { TestCriteria } { TestNotes }
                      END_OBJECTIVE ';' .
```

An OBJECTIVE declaration creates a new scope in which the following may be declared:

- The purpose of a test case;
- Reference to appropriate standards or specifications;
- Test criteria;
- Notes for the test analyst.

EXAMPLE 36 – An OBJECTIVE.

```
OBJECTIVE
  NOTES This objective only contains
        a note to the analyst.
  END_NOTES;
END_OBJECTIVE;
```

### 9.4.1 Test purpose

A test purpose is text to be read by a human. It provides a description of the intent of a test.

**Syntax:**

```
126i TestPurpose = PURPOSE Description END_PURPOSE ';' .
26i  Description = { \a | \s | \n } .
```

The text commences with the keyword PURPOSE and is terminated by the keyword END_PURPOSE and a semicolon. The text may span multiple lines.

EXAMPLE 37 – The text for this purpose extends over two lines.

```
PURPOSE This test is intended to check
        the existance of a car instance. END_PURPOSE;
```

### 9.4.2 Test reference

A test reference is text to be read by a human. It provides a description of human interpretable references to appropriate standards or specifications.

**Syntax:**

```
127i TestReference = REFERENCES Description END_REFERENCES ';' .
26i  Description = { \a | \s | \n } .
```

The text commences with the keyword REFERENCES and is terminated by the keyword END_REFERENCES and a semicolon. The text may span multiple lines.

EXAMPLE 38 – A reference to a printed document.

```
REFERENCES Document AP279, pages 53-57. END_REFERENCES;
```

### 9.4.3 Test criteria

A test criteria is text to be read by a human. It provides a description of the criteria to be used in judging the result of a test.

**Syntax:**

```
124i TestCriteria = CRITERIA Description END_CRITERIA ';' .
26i  Description = { \a | \s | \n } .
```

The text commences with the keyword CRITERIA and is terminated by the keyword END_CRITERIA and a semicolon. The text may span multiple lines.

EXAMPLE 39 – A simple criterion.

```
CRITERIA At least one instance of a car must be present. END_CRITERIA;
```

### 9.4.4 Test notes

Test notes is text to be read by a human. It provides a means of describing general notes to assist the test analyst.

**Syntax:**

```
125i TestNotes = NOTES Description END_NOTES ';' .
26i  Description = { \a | \s | \n } .
```

The text commences with the keyword NOTES and is terminated by the keyword END_NOTES and a semicolon. The text may span multiple lines.

EXAMPLE 40 – A single line note.

```
NOTES Remember to fasten your seat belt. END_NOTES;
```

## 9.5 Test realization

A test realization provides for the definition of the data elements pertaining to a test case.

**Syntax:**

```
123i TestRealization = REALIZATION { local_decl } { UseContextBlock }
                       { assignment_stmt } END_REALIZATION ';' .
239  local_decl = < as EXPRESS > .
166  assignment_stmt = < as EXPRESS > .
```

A realization commences with the keyword REALIZATION and is terminated by the keyword END_REALIZATION and a semicolon.

A test realization may contain:

- References to context data and parameters (see clause 10.3);
- Local variables (specified using *EXPRESS* syntax);
- Assignment statements (specified using *EXPRESS* syntax).

EXAMPLE 41 – This realization defines `p1` to be a variable of type `point`. It then calls for the creation of a `point` at (1,2,3), assigning the instance to the variable `p1`.

```
REALIZATION
  LOCAL
    p1 : point;
  END_LOCAL;

  p1 := point(1.0, 2.0, 3.0);
END_REALIZATION;
```

# 10 Interfaces

This clause specifies the interfaces between *EXPRESS-I* instances and *EXPRESS* models, together with the interfaces between the *EXPRESS-I* constructs.

## 10.1 Schema instance interface

**Syntax:**

```
104i SchemaInstanceBlock = SCHEMA_DATA SchemaId;
                           [ SchemaInstanceBody ] END_SCHEMA_DATA ';' .
103i SchemaId = schema_ref .
152  schema_ref = < as EXPRESS > .
```

Assuming that there is an associated *EXPRESS* (or equivantly *EXPRESS-G*) SCHEMA, then the `SchemaId` refers to the name of the *EXPRESS* SCHEMA. That is, the body of the *EXPRESS-I* schema data instance contains data instances of the definitions within the identified *EXPRESS* schema. It shall not contain data instances of definitions that are external to that *EXPRESS* schema.

NOTE – References to schemas that are defined in languages other than *EXPRESS* or *EXPRESS-G* are out of scope. However, the `SchemaId` could be considered to reference a schema that has been defined in a non-*EXPRESS* language.

## 10.2 Schema reference

A schema reference enables a particular *EXPRESS* SCHEMA to be identified together with particular definitions within that schema.

**Syntax:**

```
107i SchemaReferenceSpec = WITH schema_ref [ USING '(' resource_ref
                           { ',' resource_ref } ')' ] ';' .
152  schema_ref = < as EXPRESS > .
275  resource_ref = < as EXPRESS > .
```

The `schema_ref` following the WITH keyword identifies a particular *EXPRESS* schema. Individual declarations of interest within the *EXPRESS* schema are identified in the list following the USING keyword.

Omission of the USING list implies that all the definitions within the identified *EXPRESS* schema are available.

NOTE – The schema reference acts in a similar manner to the *EXPRESS* USE statement.

EXAMPLE 42 – Given the following *EXPRESS* definition

```
SCHEMA a_schema;
  ENTITY entity1; ... END_ENTITY;
  ENTITY entity2; ... END_ENTITY;
```

```
    ENTITY entity7; ... END_ENTITY;
    TYPE type19 = ... END_TYPE;
    TYPE type21 = ... END_TYPE;
  END_SCHEMA;
  SCHEMA another_schema;
    ...
  END_SCHEMA;
```

Then the following identifies two entities and one type from the `a_schema` schema.

```
      WITH a_schema USING (entity1, entity7, type21);
```

## 10.3 Context data references

Elements of a CONTEXT can be imported into a TEST_CASE and actual values can be given to the formal parameters in the CONTEXT.

**Syntax:**

```
134i UseContextBlock = CALL ContextRef ';' UseContextBody  END_CALL ';' .
35i  ContextRef = ContextId .
135i UseContextBody = [ ImportSpec ] [ ParameterSpec ] .
79i  ImportSpec = IMPORT '(' { Assignment } ')' ';' .
44i  Assignment = variable_id ':=' SelectableInstanceRef ';' .
96i  ParameterSpec = WITH '(' { ActualParameter } ')' ';' .
108i SelectableInstanceRef = EntityInstanceRef | EnumerationInstanceRef |
                             SelectInstanceRef | TypeInstanceRef .
```

A particular CONTEXT is identified via the CALL statement.

Object instances of interest to a test case that exist in the CONTEXT are identified in the IMPORT list. Each instance value shall be assigned to a variable.

Values for the formal parameters in the CONTEXT (if any) are set via the WITH list. These values shall overide the default value (if any) of the identified parameters.

EXAMPLE 43 – A CALL specification

```
    CALL a_context;
      IMPORT (ent_var := @ent_21;
              ent_27 := @ent_27;);
      WITH (iv1 := 771;
            a_set := ['alpha', 'to', 'omega']; );
    END_CALL;
```

# 11 Scope and visibility

An *EXPRESS-I* declaration creates an identifier which can be used to reference the declared item in other contexts. Some *EXPRESS-I* constructs implicitly declare *EXPRESS-I* items, attaching identifiers to them. In those areas where an identifier for a declared item may be referenced, the declared item is said to be visible. An item may only be referenced where its identifier is visible. For the rules of visibility see 11.2.

Certain *EXPRESS-I* items define a region (block) of text called the scope of the item. This scope limits the visibility of identifiers declared within it. Scopes can be nested; that is, an *EXPRESS-I* item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular *EXPRESS-I* item's scope. These constraints are usually enforced by the syntax of *EXPRESS-I* (see annex A).

**Table 9 – Scope and identifier defining *EXPRESS-I* items**

| Item | Scope | Identifier |
|---|---|---|
| constant instance | | ● |
| context | ● | ● |
| entity instance | | ● |
| enumeration instance | | ● |
| model | ● | ● |
| schema data instance | ● | ● |
| select instance | | ● |
| simple instance | | ● |
| test case | ● | ● |
| type instance | | ● |

NOTE – *EXPRESS-I* also utilises various *EXPRESS* constructs that similarly have identifiers and scope. These are listed in table 10.

For each of the items specified in table 9 and table 10 the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

## 11.1 Scope rules

The following are the general rules which are applicable to all forms of scope definition allowed within the *EXPRESS-I* language; see table 9 and table 10 for the list of items which define scopes.

**Rules and restrictions:**

a) All declarations shall exist within a scope.

b) Within a single scope an identifier may be declared, or explicitly interfaced, once only.

c) The scopes shall be correctly nested, i.e., scopes shall not overlap (this is forced by the syntax of the language).

A maximum permitted depth of nesting is not specified by this part of ISO 10303 but implementations of *EXPRESS-I* parsers may specify a maximum depth of scope nesting.

**Table 10 – Scope and identifier defining *EXPRESS* items utilised by *EXPRESS-I*.**

| Item | Scope | Identifier |
|---|---|---|
| alias statement | ● | ●[1] |
| attribute | | ● |
| constant | | ● |
| entity | ● | ● |
| enumeration | | ● |
| function | ● | ● |
| parameter | | ● |
| procedure | ● | ● |
| query expression | ● | ●[1] |
| repeat statement | ● | ●[1,2] |
| rule label | | ● |
| type | ● | ● |
| type label | | ● |
| variable | | ● |

NOTES

1 – The identifier is an implicitly declared variable within the defined scope of the declaration.

2 – The variable is only implicitly declared when an increment control is specified.

## 11.2 Visibility rules

The visibility rules for identifiers are described below. See table 9 and table 10 for the list of *EXPRESS-I* items which declare identifiers. The visibility rules for named data type identifiers are slightly different from those for other identifiers; these differences are described in 11.2.2.

### 11.2.1 General rules of visibility

The following are the general rules which are applicable to all identifiers except the named data type identifiers, for which rule (d) does not apply.

**Rules and restrictions:**

a) An identifier is visible in the scope in which it is declared. This scope is called the local scope of the identifier.

b) An identifier is visible in a particular scope, it is also visible in all scopes defined within that scope, subject to rule (d).

c) An identifier is not visible in any scope outside its local scope, subject to rule (f).

d) When an identifier $i$ visible in a scope $P$ is re-declared in some inner scope $Q$ enclosed within $P$, only the $i$ declared in scope $Q$ is visible in $Q$ and any scopes declared within $Q$. The $i$ declared in scope $P$ is visible in $P$ and in any inner scopes which do not re-declare $i$.

e) The built-in constants, functions, procedures and types of *EXPRESS-I* are considered to be declared in an imaginary universal scope. All *EXPRESS-I* scopes are nested within this scope. The identifiers which refer to the built-in constants, functions, procedures and types of *EXPRESS-I* are visible in all scopes defined by *EXPRESS-I*.

f) Enumeration item identifiers declared within the scope of a defined data type are visible in the next outer scope, unless the next outer scope contains a declaration of the same identifier for another item.

NOTE – If the next outer scope contains a declaration of the same identifier, the enumeration items are still accessible but have to be prefixed by the defined data type identifier.

g) Some *EXPRESS-I* declarations which are normally invisible may be made visible by interface specifications (see clause 10).

### 11.2.2 Named data type identifier visibility rules

With one exception, named data type identifiers obey the same visibility rules as other identifiers. The exception is to visibility rule (d). An entity or defined data type identifier $i$ declared in a scope $P$ remains visible in an inner scope $Q$ even if it is redeclared in $Q$, provided that either:

a) The scope $Q$ is defined by an entity declaration, and $i$ is declared as an attribute in that scope, or

b) The scope $Q$ is defined by a function, procedure or context declaration, and $i$ is declared as a formal parameter or variable in that scope.

EXAMPLE 44 – In `entity1`, `d` refers to both an entity data type and an attribute.

```
FUNCTION example(par : INTEGER): INTEGER;

  ENTITY d;
    attr1 : REAL;
  END_ENTITY;

  ENTITY entity1;
    d : d;                    -- d in this scope is both an entity
  END_ENTITY;                 --  and an attribute.
  ...
  ...
END_FUNCTION;
```

## 11.3 Explicit item rules

The following clauses provide more detail on how the general scoping and visibility rules apply to the various *EXPRESS-I* items.

*EXPRESS-I* utilises much of the *EXPRESS* language. The scoping and visibility rules for most of these *EXPRESS* items within *EXPRESS-I* are identical to those of *EXPRESS* as defined in ISO 10303. Table 11 identifies these items. The table further identifies those items common to both *EXPRESS* and *EXPRESS-I* whose *EXPRESS* rules are modified when they are used within *EXPRESS-I* and those items which are particular to *EXPRESS-I*.

NOTE – The modifications to the *EXPRESS* rules are due principally to the fact that *EXPRESS-I* does not utilise the *EXPRESS* SCHEMA or RULE constructs.

### 11.3.1 Alias statement

The scope and visibilty rules for the ALIAS statement are defined in ISO 10303-11.

### 11.3.2 Attribute

The scope and visibilty rules for an attribute are defined in ISO 10303-11.

### 11.3.3 Constant

**Visibility:** A constant identifier is visible in the scope of the function or procedure in which it is declared.

NOTE – The *EXPRESS* specification is:

**Table 11 – Scope and visibility rules.**

| Item | *EXPRESS* rules | *EXPRESS* modified rules | *EXPRESS-I* specific |
|---|---|---|---|
| alias statement | ● | | |
| attribute | ● | | |
| constant | | ● | |
| constant instance | | | ● |
| context | | | ● |
| entity | | ● | |
| entity instance | | | ● |
| enumeration | | ● | |
| enumeration instance | | | ● |
| function | | ● | |
| model | | | ● |
| parameter | | ● | |
| procedure | | ● | |
| query expression | ● | | |
| repeat statement | ● | | |
| rule label | | ● | |
| schema data instance | | | ● |
| select instance | | | ● |
| simple instance | | | ● |
| test case | | | ● |
| type | | ● | |
| type instance | | | ● |
| type label | ● | | |
| variable | | ● | |

> A constant identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

### 11.3.4 Constant instance

**Visibility:** A constance instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.5 Context

**Visibility:** A context identifier is visible to all test cases.

**Scope:** A context declaration defines a new scope. The keyword CONTEXT starts this scope which extends to the keyword END_CONTEXT which terminates that context declaration.

**Declarations:** The following items may declare identifiers within the scope of a context declaration:

- formal parameter;
- function;
- procedure;
- schema data instance.

### 11.3.6 Entity

**Visibility:** An entity identifier is visible in the scope of the function or procedure in which it is declared. An entity identifier remains visible, under the conditions defined in 11.2.2, within inner scopes which redeclare that identifier.

NOTE – The *EXPRESS* specification is:

> An entity identifier is visible in the scope of the function, procedure, rule or schema in which it is declared. An entity identifier remains visible ...

**Scope and declarations:** The scope and allowable declarations are defined in ISO 10303-11.

EXAMPLE 45 – The attribute identifiers `batt` in the two entities do not clash as they are declared in two different scopes.

```
ENTITY entity1;
  aatt : INTEGER;
  batt : INTEGER;
```

```
END_ENTITY;

ENTITY entity2;
  a    : entity1;
  batt : INTEGER;
END_ENTITY;
```

EXAMPLE 46 – The following specification is illegal because the attribute identifier `aatt` is repeated within the scope of a single entity. Although the rule label `lab` is declared in both entities, this does not violate any scoping or visibility rule; the declaration in entity `may_be_ok` is not visible in the entity `illegal`, but both domain rules must be checked.

```
ENTITY may_be_ok;
  quantity : REAL;
WHERE
  lab : quantity >= 0.0;
END_ENTITY;

ENTITY illegal
  SUBTYPE OF (may_be_ok);
  aatt : INTEGER;
  batt : INTEGER;
  aatt : REAL;
WHERE
  lab : batt < 0;
END_ENTITY;
```

### 11.3.7 Entity instance

**Visibility:** An entity instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.8 Enumeration item

**Visibility:** An enumeration item identifier is visible in the scope of the function or procedure in which its type is declared. This is the exception to the visibility rule f of 11.2.1. The identifier shall not be declared for any other purpose in this scope, except by another enumeration data type declaration in the same scope. If the same identifier is declared by two enumeration data types as an enumeration item, a reference to either enumeration item shall be prefixed with the data type identifier in order to ensure that the reference is unambiguous.

NOTE – The *EXPRESS* specification is:

> An enumeration item identifier is visible in the scope of the function, procedure, rule or schema in which its type is declared. This is the exception ...

### 11.3.9 Enumeration instance

**Visibility:** An enumeration instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.10 Function

**Visibility:** A function identifier is visible in the scope of the function, procedure, context or test case in which it is declared.

NOTE – The *EXPRESS* specification is:

> A function identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

**Scope and declarations:** The scope and allowable declarations are defined in ISO 10303-11.

### 11.3.11 Model

**Scope:** A model declaration defines a new scope. This scope extends from the keyword MODEL to the keyword END_MODEL which terminates that model declaration.

**Declarations:** The following items may declare identifiers within the scope of a model declaration:

– schema data instance.

### 11.3.12 Parameter

**Visibility:** A formal parameter identifier is visible in the scope of the function, procedure or context in which it is declared.

NOTE – The *EXPRESS* specification is:

> A formal parameter identifier is visible in the scope of the function or procedure in which it is declared.

EXAMPLE 47 – The following is illegal, as the formal parameter identifier `parm` is also used as the identifier of a local variable.

```
CONTEXT illegal;
  PARAMETER
    parm : REAL;
    ...
  END_PARAMETER;
```

```
    LOCAL
      parm : STRING;
    END_LOCAL;
    ...
  END_CONTEXT;
```

### 11.3.13 Procedure

**Visibility:** A procedure identifier is visible in the scope of the function, procedure, context or test case in which it is declared.

NOTE – The *EXPRESS* specification is:

> A procedure identifier is visible in the scope of the function, procedure, rule or schema in which it is declared.

**Scope and declarations:** The scope and allowable declarations are defined in ISO 10303-11.

### 11.3.14 Query expression

The scope and visibility of a QUERY expression is defined in ISO 10303-11.

### 11.3.15 Repeat statement

The scope and visibility of a REPEAT statement is defined in ISO 10303-11.

### 11.3.16 Rule label

**Visibility:** A rule label is visible in the scope of the entity or type in which it is declared.

NOTE 1 – The *EXPRESS* specification is:

> A rule label is visible in the scope of the entity, rule or type in which it is declared.

NOTE 2 – The rule label is only of use to an implementation. Neither *EXPRESS* nor *EXPRESS-I* provides a mechanism for referencing rule labels.

### 11.3.17 Schema data instance

**Scope:** A schema data declaration defines a new scope. This scope extends from the keyword SCHEMA_DATA to the keyword END_SCHEMA_DATA which terminates that schema data declaration.

**Declarations:** The following items may declare identifiers within the scope of a schema data declaration:

- constant instance;
- entity instance;
- enumeration instance;
- select instance;
- simple instance;
- type instance.

### 11.3.18 Select instance

**Visibility:** A select instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.19 Simple instance

**Visibility:** A simple instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.20 Test case

**Scope:** A test case defines a new scope. This scope extends from the keyword TEST_CASE to the keyword END_TEST_CASE which terminates that test case.

**Declarations:** The following items may declare identifiers within the scope of a test case:

- function;
- procedure;
- variable.

### 11.3.21 Type

**Visibility:** A type identifier is visible in the scope of the function or procedure in which it is declared. A type identifier remains visible, under certain conditions, in inner scopes which redeclare that identifier; see 11.2.2 for the definition of the allowed conditions.

NOTE – The *EXPRESS* specification is:

> A type identifier is visible in the scope of the function, procedure, rule or schema in which it is declared. A type identifier remains visible ...

**Scope and declarations:** The scope and allowable declarations are defined in ISO 10303-11.

### 11.3.22 Type instance

**Visibility:** A type instance identifier is visible in the scope of the schema data instance in which it is declared and in any outer scope of the schema data instance.

### 11.3.23 Type label

The scope and visibility are defined in ISO 10303-11.

### 11.3.24 Variable

**Visibility:** A variable identifier is visible in the scope of the function, procedure or test case in which it is declared.

NOTE – The *EXPRESS* specification is:

> A variable identifier is visible in the scope of the function, procedure or rule in which it is declared.

## 12 Mapping from EXPRESS to EXPRESS-I

This clause specifies the mapping of *EXPRESS* schema and type definitions to *EXPRESS-I* instances.

Table 12 gives an overview of the *EXPRESS* to *EXPRESS-I* mappings. These are described in more detail below.

### 12.1 Mapping of EXPRESS schema

The *EXPRESS* construct of SCHEMA maps syntactically to the *EXPRESS-I* construct of schema data instance. Table 13 gives an overview of the correspondance between the *EXPRESS* and *EXPRESS-I* constructs.

**Rules and restrictions:**

a) The name of the *EXPRESS-I* schema data instance shall be the same as the name of the corresponding *EXPRESS* schema.

b) Each entity instance within a schema data instance shall have a corresponding entity definition within the *EXPRESS* schema.

**Table 12 – Summary overview of *EXPRESS* to *EXPRESS-I* mappings.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| ARRAY, BAG, LIST, SET | AggregationValue |
| CONSTANT | ConstantBlock |
| | ContextBlock |
| ENTITY | EntityInstance |
| ENUMERATION | Enumeration instance or value |
| | FormalParameterBlock |
| FUNCTION | |
| | ModelBlock |
| PROCEDURE | |
| Remark | |
| RULE | |
| SCHEMA | SchemaInstanceBlock |
| SELECT | Select instance or value |
| Simple type | SimpleValue |
| | TestCaseBlock |
| TYPE | Type instance or value |

**Table 13 – Overview of SCHEMA mapping.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| SCHEMA name | schema_id |
| CONSTANT | ConstantBlock or none |
| ENTITY | EntityInstance |
| ENUMERATION | EnumerationInstance or none |
| FUNCTION | none |
| PROCEDURE | none |
| REFERENCE | none, but see clause 12.1.1 |
| RULE | none |
| SELECT | SelectInstance or none |
| TYPE | TypeInstance or none |
| USE | none, but see clause 12.1.1 |

c) Each enumeration, select or type instance within a schema data instance shall have a corresponding definition within the *EXPRESS* schema.

d) Each constant within a schema data instance shall have a corresponding constant definition within the *EXPRESS* schema.

e) Each domain specification within a schema data instance shall be uniquely identified, if necessary by qualifying the domain name with the name of the *EXPRESS* schema which contains the domain definition.

f) Instance identifiers shall be unique within a schema data instance.

### 12.1.1 Mapping of use and reference

The *EXPRESS* USE and REFERENCE statements do not map directly to *EXPRESS-I* but their effects do occur:

– Instances of *EXPRESS* elements that are brought within the scope of an *EXPRESS* schema via explicit USE or REFERENCE statements, or that are implicitly referenced, may occur within a corresponding *EXPRESS-I* schema data instance.

– Elements whose domains are renamed, shall have their domains specified via the new names.

– If there are name clashes between the domains in the original *EXPRESS* schema and those that are brought in from another schema, then the brought in names shall be qualified with the name of their parent schema.

EXAMPLE 48 – These *EXPRESS* schemas are interlinked as the schema called `primary` utilizes the definition of the entity called `an_ent` from the `secondary` schema.

```
SCHEMA primary;
  USE FROM secondary (an_ent AS used);

  ENTITY dup;
    att1 : used;
    att2 : BOOLEAN;
  END_ENTITY;
END_SCHEMA;

SCHEMA secondary;

  ENTITY dup;
    name : STRING;
    int  : INTEGER;
  END_ENTITY;

  ENTITY an_ent;
    att3 : dup;
    att4 : REAL;
  END_ENTITY;
END_SCHEMA;
```

Any usage of `an_ent` in an instance of the `primary` schema requires an instance of the entity called `dup` which is also defined in the `secondary` schema and which is automatically made available through the semantics of the USE clause. However, in this case, there is also an entity called `dup` in the `primary` schema. These two domains must be distinguished within an *EXPRESS-I* representation of `primary` by qualifying the name of the enity that is brought in from the `secondary` schema, as in the following.

```
MODEL example;
  SCHEMA_DATA primary;
    dup1 = dup{att1 -> @used1; att2 -> TRUE;};
    used1 = used{att3 -> @dup2; att4 -> 1.23;};
    dup2 = secondary.dup{name -> 'from secondary'; int -> 1;};
    used2 = used{att3 -> @dup3; att4 -> -3.9;};
    dup3 = secondary.dup{name -> 'from secondary'; int -> 2;};
  END_SCHEMA_DATA;

  SCHEMA_DATA secondary;
    dup3 = dup{name -> 'in secondary'; int -> 3;};
    dup4 = dup{name -> 'in secondary'; int -> 4;};
    an_ent1 = an_ent{att3 -> @dup3; att4 -> 42.0;};
  END_SCHEMA_DATA;
END_MODEL;
```

## 12.2 Mapping of EXPRESS simple data types

The mapping from an *EXPRESS* simple data type to an *EXPRESS-I* value is given in table 14.

**Table 14 – Simple type mapping.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| BINARY | `BinaryValue` |
| BOOLEAN | `BooleanValue` |
| INTEGER | `IntegerValue` |
| LOGICAL | `LogicalValue` |
| NUMBER | `IntegerValue`<br>`SignedMathConstant`<br>`SignedRealValue` |
| REAL | `SignedMathConstant`<br>`SignedRealValue` |
| STRING | `StringValue` |

EXAMPLE 49 – Mapping of simple data types

```
    EXPRESS                                  EXPRESS-I
    =======                                  =========
ENTITY base;                          e1 = base{
    a_binary    : BINARY;                                a_binary   -> %0110;
    a_boolean   : BOOLEAN;                               a_boolean  -> FALSE;
    an_integer  : INTEGER;                               an_integer -> 12345;
    a_logical   : LOGICAL;                               a_logical  -> UNKNOWN;
    a_number    : NUMBER;                                a_number   -> -PI;
    a_real      : REAL;                                  a_real     -> -9.99e2;
    a_string    : STRING;                                a_string   -> 'Tangles';
  END_ENTITY;                                          };
```

## 12.3 Mapping of aggregation data types

The mapping of *EXPRESS* aggregations to *EXPRESS-I* is given in table 15.

**Table 15 – Mapping of AGGREGATEs.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| AGGREGATE | one of the following: |
| ARRAY | `FixedAggr` |
| BAG | `DynamicAggr` |
| LIST | `DynamicAggr` |
| SET | `DynamicAggr` |

The mapping of "aggregation of aggregation of ..." is done by mapping each elemental aggregation in order, reading from left to right. That is, the leftmost *EXPRESS* aggregation becomes the outermost *EXPRESS-I* aggregation.

EXAMPLE 50 – Aggregate mappings

```
    EXPRESS                                        EXPRESS-I
    =======                                        =========
ENTITY aggr;                                   e1 = aggr{
 an_array : ARRAY [1:3] OF INTEGER;                  an_array -> [1,2,3];
 a_bag    : BAG [0:?] OF INTEGER;                    a_bag -> (3,3,1);
 a_list   : LIST [0:2] OF INTEGER;                   a_list -> (1);
 a_set    : SET [1:?] OF INTEGER;                    a_set -> (9,5,11);
 a_mix    : ARRAY [1:2] OF SET OF INTEGER;           a_mix -> [(1,2),(6,5)];
END_ENTITY;                                          };
```

NOTE – An *EXPRESS* ARRAY may have OPTIONAL values. If the values are unspecified in an instance of an ARRAY then these values are denoted by the `Nil` construct (i.e the `?` character) in *EXPRESS-I*.

EXAMPLE 51 – Sparse array mapping

```
    EXPRESS                                        EXPRESS-I
    =======                                        =========
ENTITY sparse;                                   e1 = sparse{
  a1 : ARRAY [1:4] OF OPTIONAL INTEGER;                     a1 -> [1,?,?,4];
  a2 : ARRAY [5:8] OF OPTIONAL INTEGER;                     a2 -> [1,?,3,?];
END_ENTITY;                                                 };
```

## 12.4 Mapping of EXPRESS defined data type

An *EXPRESS* defined data type is mapped to *EXPRESS-I* in one of three ways:

a) by replacing the *EXPRESS* type identifier by the type value;

b) by replacing the *EXPRESS* type identifier by the named type value;

c) by specifying a type instance.

EXAMPLE 52 – Mapping a defined data type

```
    EXPRESS                                EXPRESS-I
    =======                                =========
TYPE dd = ARRAY [1:2] OF          t3 = dd{[6,8]};
          INTEGER;
END_TYPE;

ENTITY use_type;                  e1 = use_type{attr -> [2,4];};
  attr : dd;                      e2 = use_type{attr -> dd{[4,6]};};
END_ENTITY;                       e3 = use_type{attr -> @t3;};
```

## 12.5 Mapping of EXPRESS enumeration type

An *EXPRESS* ENUMERATION type is mapped to *EXPRESS-I* in one of three ways:

a) by replacing the *EXPRESS* type identifier by the enumeration value;

b) by replacing the *EXPRESS* type identifier by the named enumeration value;

c) by specifying an enumeration instance.

EXAMPLE 53 – Mapping an enumeration

```
    EXPRESS                                EXPRESS-I
    =======                                =========
TYPE enum = ENUMERATION OF        t3 = enum{!three};
     (one, two, three);
END_TYPE;

ENTITY use_enum;                  e1 = use_enum{attr -> !one;};
  attr : enum;                    e2 = use_enum{attr -> enum{!two};};
END_ENTITY;                       e3 = use_enum{attr -> @t3;};
```

## 12.6 Mapping of EXPRESS select type

An *EXPRESS* SELECT type is mapped to *EXPRESS-I* in one of three ways:

a) by replacing the *EXPRESS* type identifier by the select value;

b) by replacing the *EXPRESS* type identifier by the named select value;

c) by specifying a select instance.

An *EXPRESS* SELECT type may not necessarily be mapped directly into *EXPRESS-I*. The details of the mapping depend on how the SELECT type is formed, as described below.

A SELECT type defines a tree. The root is the SELECT type and the branches from the root correspond to the types of the choices within the SELECT. If one of these types is itself a SELECT then this gives rise to further branches, and so on. The leaves of the tree are composed of the choices that are not SELECT types. In the simple case all leaves are of different types. In the complex case, at least two of the leaves have the same base type.

### 12.6.1 Simple select case

The type is treated as a reference to, or an occurrence of, one of the types in its select list.

EXAMPLE 54 – Simple select mapping

```
    EXPRESS                                  EXPRESS-I
    =======                                  =========
ENTITY a;                              e1 = a{aa -> 3;};
  aa : INTEGER;                        e3 = a{aa -> 9;};
END_ENTITY;

ENTITY b;                              e2 = b{ab -> 6;};
  ab : INTEGER;                        e4 = b{ab -> 12;};
END_ENTITY;

TYPE s = SELECT (a, b);                s4 = s{@e4};
END_TYPE;

ENTITY c;                              c1 = c{ac -> (@s4, @3, @2, @1);};
  ac : LIST [1:?] OF s;                c2 = c{ac -> (s{@1}, @3, @3);};
END_ENTITY;
```

### 12.6.2 Complex select case

In this case, the leaves of the tree are not distinguishable by their value alone. This occurs when:

a) the leaves are defined data types with identical base types, or

b) the leaves are ENUMERATION types where the set of values in the leaves are not disjoint. For example, the sets `[red, green, blue]` and `[red, amber, green]` are not disjoint.

The value of the select instance in this case shall be represented in *EXPRESS-I* either by a reference to an instance or by a named value.

EXAMPLE 55 – Complex select mapping

```
    EXPRESS                                  EXPRESS-I
    =======                                  =========
TYPE size = SELECT                   s1 = size{@r1};
            (area, radius);          s2 = size{radius{4.3}};
```

```
END_TYPE;

TYPE area = REAL;                   a1 = area{7.5};
END_TYPE;

TYPE radius = REAL;                 r1 = radius{27.89};
END_TYPE;

ENTITY circle;                      c1 = circle{howbig -> area{PI};};
  howbig : size;                    c2 = circle{howbig -> radius{1.0};};
WHERE                               c3 = circle{howbig -> @s1;};
  howbig > 0.0;                     c4 = circle{howbig -> @a1};
END_ENTITY;                         c5 = circle{howbig -> @s2};
```

## 12.7 Mapping of EXPRESS constant

An *EXPRESS* CONSTANT maps syntactically to the *EXPRESS-I* construct of `constant_spec`. That is, the constant identifier and value only is specified in *EXPRESS-I* — the domain of the constant value is provided by the original *EXPRESS* definition. Further, the constant value shall be completely evaluated. Each constant specification appearing in a schema instance shall have been declared in the *EXPRESS* schema definition. However, it is not required that each *EXPRESS* CONSTANT appear within a schema instance.

EXAMPLE 56 – Constant mapping

```
    EXPRESS                                  EXPRESS-I
    =======                                  =========
CONSTANT                                 CONSTANT
  zero : NUMBER := 0.0;                    zero == 0.0;
  thousand : INTEGER := 1000;              thousand == 1000;
  million : INTEGER := thousand**2;        million == 1000000;
  origin : point := point(0.0, 0.0);       origin == point{x -> 0.0;
                                                           y -> 0.0;};
  z_axis : vector := [zero, zero, 1.0];    z_axis == [0.0, 0.0, 1.0];
  a_set : SET OF INTEGER := [1,2,3*3];     a_set == (1, 2, 9);
  a_bag : BAG OF INTEGER := [1,3,1];
  boss : STRING := 'sir' ;
  underling : STRING := 'hey, you';        underling == 'hey, you';
END_CONSTANT;                            END_CONSTANT;
```

## 12.8 Mapping of EXPRESS entity

The *EXPRESS* construct of ENTITY maps syntactically to the *EXPRESS-I* construct of entity instance. It is to be noted that the only internal portions of an ENTITY that are mapped to *EXPRESS-I* are attributes, and SUPERTYPE and SUBTYPE clauses, as listed in table 16.

EXAMPLE 57 – Simple entity mapping

```
    EXPRESS                                  EXPRESS-I
    =======                                  =========
```

**Table 16 – Overview of ENTITY mapping.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| ENTITY name | `EntityDomain` |
| SUPERTYPE clause | `BequeathsTo` |
| SUBTYPE clause | `InheritsFrom` |
| explicit attribute | `RequiredAttr` or `OptionalAttr` |
| derived attribute | `DerivedAttr` |
| inverse attribute | `InverseAttr` |
| UNIQUE clause | none |
| WHERE clause | none |

```
ENTITY top;                             t1 = top{a -> (@eg1, @eg2);};
  a : SET OF bot;                       t2 = top{a -> (@eg2, @eg3);};
END_ENTITY;                             t3 = top{a -> ();};

ENTITY bot;                             eg1 = bot{i -> 1;
  i : INTEGER;                                    j <- 2;
DERIVE                                            inv <- (@t1);};
  j : INTEGER := 2*i;
INVERSE                                 eg2 = bot{i -> 276;
  inv : BAG [1:?] OF top FOR attr;                j <- 552;
UNIQUE                                            inv <- (@t1, @t2);};
  u1 : i;
WHERE                                   eg3 = bot{i -> 9876;
  w1 : i > 0;                                     j;
END_ENTITY;                                       inv <- (@t2);};
```

## 12.9 Mapping of EXPRESS entity attributes

*EXPRESS-I* attributes shall appear in the same order as in the corresponding *EXPRESS* ENTITY. Each *EXPRESS* attribute shall have a corresponding *EXPRESS-I* attribute.

The *EXPRESS-I* value of an attribute shall be compatible with the domain of the *EXPRESS* definition.

### 12.9.1 Explicit attribute

Explicit *EXPRESS* attributes map in a straightforward manner to *EXPRESS-I* attributes. The description of the *EXPRESS* attribute is repeated in *EXPRESS-I* except that the description of the type of the attribute (i.e the right hand side after the colon) is replaced by the value of the attribute type and the colon is replaced by `->`.

The value may be represented by a simple value, an object instance reference (i.e an entity, type, enumeration or select instance reference), an enumeration value, a named value, a constant reference, or a parameter reference, or aggregates of these. These are discussed in more detail below.

In the case where an explict attribute is OPTIONAL the attribute value may also be `Nil`, indicating that the value is not supplied.

EXAMPLE 58 – Mapping an optional attribute

```
    EXPRESS                                   EXPRESS-I
    =======                                   =========
ENTITY opt;                     opt1 = opt{req -> 'Opt-att given';
  req     : STRING;                       opt_att -> 5.0; };
  opt_att : OPTIONAL REAL;
END_ENTITY;                     opt2 = opt{req -> 'Opt-att not given';
                                          opt_att -> ?; };
```

NOTE – In *EXPRESS-I* a non-optional explicit attribute may have a `Nil` value, in which case the instance is non-conforming with respect to the *EXPRESS* definition.

### 12.9.2 Derived and inverse attributes

Derived *EXPRESS* attributes map to *EXPRESS-I* in a similar manner to explicit attributes, except that the symbol `<-` relaces the colon.

Inverse *EXPRESS* attributes map to *EXPRESS-I* in a similar manner to explicit attributes, except that the symbol `<-` relaces the colon, and the attribute value is a dynamic aggregation of entity instance references.

It should be noted that there is no requirement that the values of derived or inverse attributes appear in *EXPRESS-I* although the role names shall appear.

NOTES

1 – By definition, the value of a derived attribute can be determined from the values of the explicit attributes. Similarly, the value of an inverse attribute of an entity instance can be determined from the attribute values of other entity instances that reference the entity instance with the given inverse attribute. Thus, conceptually at least, both derived and inverse attribute values are calculable properties.

2 – On the other hand, the values of explicit attributes are basic input data that is not calculable within an *EXPRESS-I* system.

3 – The symbols `->` and `<-` were designed to indicate this difference in the qualities of attribute values.

### 12.9.3 Attribute with a simple domain

When the domain of an *EXPRESS* attribute is a simple data type this shall be mapped as an *EXPRESS-I* value belonging the simple domain. Typically this is a simple value, but may be a constant or parameter reference whose domain is the simple domain.

**Rules and restrictions:**

a) Constant reference shall only be used if both the entity instance and the constant instance is within the the same schema data instance.

b) Parameter reference shall only be used if the formal parameter and the entity instance are both within the same CONTEXT.

c) Parameter reference shall not be used within the scope of a MODEL.

EXAMPLE 59 – Mapping a simple value as attribute:

Given the *EXPRESS* as

```
SCHEMA a_schema;
  CONSTANT
    const : INTEGER := 275;
  END_CONSTANT;

  ENTITY an_ent;
    aa : INTEGER;
  END_ENTITY;
END_SCHEMA;
```

then an *EXPRESS-I* rendition could look like:

```
MODEL some_data;

  SCHEMA_DATA a_schema;

    CONSTANT
      const == 275;
    END_CONSTANT;

    a1 = an_ent{aa -> 1;};
    a2 = an_ent{aa -> const;};
    a3 = an_ent{aa -> 21;};
    a4 = an_ent{aa -> 987;};
  END_SCHEMA_DATA;
END_MODEL;
```

Alternatively, it could be represented via a context as:

```
CONTEXT a_context;
  PARAMETER
    param1 : INTEGER := 21;
    param2 : INTEGER := 987;
  END_PARAMETER;

  SCHEMA_DATA a_schema;

    CONSTANT
      const == 275;
    END_CONSTANT;

    a1 = an_ent{aa -> 1;};
    a2 = an_ent{aa -> const};
    a3 = an_ent{aa -> param1};
    a4 = an_ent{aa -> param2};
  END_SCHEMA_DATA;
END_CONTEXT;
```

### 12.9.4 Attribute with an entity domain

When the domain of an *EXPRESS* attribute is an entity, this shall be mapped as an *EXPRESS-I* value belonging the entity domain. Typically this is an entity instance reference, but may be a constant or parameter reference whose domain is the entity domain.

**Rules and restrictions:**

a) Constant reference shall only be used if both the entity instance and the constant instance is within the the same schema data instance.

b) Parameter reference shall only be used if the formal parameter and the entity instance are both within the same CONTEXT.

c) Parameter reference shall not be used within the scope of a MODEL.

d) Neither parameter nor constant reference shall be used for an inverse attribute.

EXAMPLE 60 – Mapping an entity as attribute:

Given the *EXPRESS* as

```
SCHEMA a_schema;
  CONSTANT
    const : an_ent := an_ent(275);
  END_CONSTANT;

  ENTITY an_ent;
    aa : INTEGER;
  END_ENTITY;

  ENTITY bdyn;
    ab : an_ent;
  END_ENTITY;
END_SCHEMA;
```

then an *EXPRESS-I* rendition could look like:

```
CONTEXT a_context;
  PARAMETER
    param : an_ent := an_ent{aa -> 42;};
  END_PARAMETER;

  SCHEMA_DATA a_schema;

    CONSTANT
```

```
      const == an_ent{aa -> 275;};
    END_CONSTANT;

    a1 = an_ent{aa -> 1;};
    b1 = bdyn{ab -> @a1;};
    b2 = bdyn{ab -> const;};
    b3 = bdyn{ab -> param;};
  END_SCHEMA_DATA;
END_CONTEXT;
```

## 12.9.5 Attribute with a type, select or enumeration domain

When the domain of an *EXPRESS* attribute is a defined data type, a SELECT, or an ENUMERATION, this shall be mapped as an *EXPRESS-I* value belonging the domain. Typically this is a either a value (for a defined data type or enumeration) or an entity instance reference (for a select), but may be an object instance reference, a named value, or a constant or parameter reference whose domain is compatible with the attribute domain.

**Rules and restrictions:**

a) Constant reference shall only be used if both the entity instance and the constant instance is within the the same schema data instance.

b) Parameter reference shall only be used if the formal parameter and the entity instance are both within the same CONTEXT.

c) Parameter reference shall not be used within the scope of a MODEL.

d) An object instance reference or a named value shall be used when the actual domain is not unambiguously determinable from the value.

EXAMPLE 61 – Mapping types as attribute:

Given the *EXPRESS* as

```
SCHEMA a_schema;
  CONSTANT
    zero : REAL := 0.0;
  END_CONSTANT;

  TYPE size = SELECT(area, radius); END_TYPE;
  TYPE area = REAL; END_TYPE;
  TYPE radius = REAL; END_TYPE;
  TYPE vector = ARRAY [1:3] OF REAL; END_TYPE;
  TYPE color = ENUMERATION OF (red, blue, green); END_TYPE;

  ENTITY point;
    x, y, z : REAL;
  END_ENTITY;

  ENTITY circle;
```

```
    center : point;
    normal : vector;
    howbig : size;
    shade  : color;
  END_ENTITY;
END_SCHEMA;
```

then an *EXPRESS-I* rendition could look like:

```
SCHEMA_DATA a_schema;

  CONSTANT
      zero == 0.0;
  END_CONSTANT;

  unit_rad = size{radius{1.0}};
  x_axis = vector{[1.0, zero, zero]};
  z_axis = vector{[zero, zero, 1.0]};
  x_color = color{"red"};

  p0 = point{x -> zero; y -> zero; z -> zero;};
  p1 = point{x -> 1.0; y -> 1.0; z -> 1.0};

  c1 = circle{center -> @p0;
              normal -> @x_axis;
              howbig -> area{PI};
              shade  -> @x_color;};
  c2 = circle{center -> @p0;
              normal -> [1.0, 2.0, 3.0];
              howbig -> radius{33.0};
              shade  -> "blue";};
  c3 = circle{center -> @p1;
              normal -> @z_axis;
              howbig -> @unit_rad;
              shade  -> "blue";};
END_SCHEMA_DATA;
```

## 12.10 Mapping of supertypes and subtypes

As table 17 shows, there is a one-to-one correspondence between the *EXPRESS* and *EXPRESS-I* super- and sub-typing.

**Table 17 – Overview of SUPERTYPE and SUBTYPE mapping.**

| *EXPRESS* | *EXPRESS-I* |
|---|---|
| SUPERTYPE OF ( ...) | `BequeathsTo` |
| SUBTYPE OF ( ...) | `InheritsFrom` |

In *EXPRESS-I* the instantiation of an entity that is the leaf of a super/subtype tree requires the instantiation of all its supertypes. An *EXPRESS-I* supertype instance tree shall always be written out in full.

NOTE – For discussion purposes, consider the portion of the *EXPRESS* tree below, and in particular the entity `me`:

```
ENTITY .....
ENTITY parent SUBTYPE OF (grandparent)
              SUPERTYPE OF (me ANDOR sibling);
              .....
ENTITY me     SUBTYPE OF (parent)
              SUPERTYPE OF (elder ANDOR younger);
              .....
ENTITY elder  SUBTYPE OF (me)
              SUPERTYPE OF .....
ENTITY .....
```

`Me` inherits any attributes that its supertypes (e.g `parent`, `grandparent` etc) may have. In turn, `me` bequeathes both its inherited attributes and its own attributes to its subtypes (e.g `elder`, `younger` and their offspring in turn).

In this tree, an instance of `me` may or may not also have a `sibling`. In a general tree there may be many relations existing that are not in the direct line of ancestry and descent.

For the purposes of this clause, define:

**Direct tree instance:** An instance of a singly rooted sub/supertype tree where there is a single direct path, with no branches, from the root to a single leaf.

**General tree instance:** An instance of a sub/supertype tree which is not a direct tree instance.

An *EXPRESS* tree where all SUPERTYPE relations are ONEOF and no SUBTYPE has multiple SUPERTYPEs is always a direct tree.

An instantiation of a tree that includes ANDOR relations will be direct if all the ANDOR relations are instantiated as ONEOF relations, otherwise at least some part of the instantiated tree will not be direct. An instantiation of an AND relation always gives a general tree. An instantiation of an ENTITY that has multiple SUPERTYPEs always gives a general tree.

In a direct tree instance the full instance path from root to leaf shall be represented.

The following set of rules specify the general tree mapping.

a) The full instance path from root to leaf, including side branches, shall always be instantiated, according to the rules below.

b) If an instantiated ENTITY is a SUBTYPE of one or more entities, then each of the SUPERTYPE entities shall be instantiated.

c) If an ENTITY is the SUPERTYPE of one or more entities (i.e there is an AND relationship or there is an ANDOR relationship which is instantiated as an AND rather than as a ONEOF relationship) then the SUPERTYPE and all its simultaneously extant SUBTYPE entities shall be instantiated.

d) If a SUPERTYPE ENTITY is marked as ABSTRACT then an instance of this entity will always have at least one instance of a SUBTYPE. If the SUPERTYPE is not marked as ABSTRACT then it may or may not have SUBTYPE instances, depending on the specific data.

NOTE 1 – The ordering of entity instances in a sub/supertype tree instance is not significant.

EXAMPLE 62 – Tree mapping

Given the following *EXPRESS* code

```
ENTITY root
  g_name : STRING;
END_ENTITY;

ENTITY node
  SUBTYPE OF (grandparent);
  p_name : STRING;
END_ENTITY;

ENTITY leaf1
  SUBTYPE OF (parent);
  my_name : STRING;
END_ENTITY;

ENTITY leaf2
  SUBTYPE OF (parent)
  s_name : STRING;
END_ENTITY;
```

then two example instances of this structure could be:

```
    INSTANCE 1                              INSTANCE 2
    ==========                              ==========
g1 = root{                            g2 = root{
        g_name -> 'Gran';                     g_name -> 'Gramps';
        SUPOF(@p1);};                         SUPOF(@p2);};

p1 = node{                            p2 = node{
        SUBOF(@g1);                           SUBOF(@g2);
        p_name -> 'Dad';                      p_name -> 'Mum";
        SUPOF(@c1,@s1);};                     SUPOF(@c2);};

c1 = leaf1{                           c2 = leaf1{
        SUBOF(@p1);                           SUBOF(@p2);
        my_name -> 'self';};                  my_name -> 'ego';};

s1 = leaf2{
        SUBOF(@p1);
        s_name -> 'Sis';};
```

The instance labelled 1 is a general tree instance and the one labelled 2 is a direct tree instance.

## 12.10.1 Mapping of redeclared attributes

In an *EXPRESS* subtype it is possible to redeclare attributes that are inherited from a supertype. In *EXPRESS-I* the redeclaration is treated as a constraint on the value of the attribute. Redeclared attributes shall not be be named within an instance of the subtype.

EXAMPLE 63 – In the following the entity `real_point` is a subtype of `point` and redeclares its attributes to be of type REAL instead of type NUMBER. Their are two corresponding *EXPRESS-I* instances. The first instance (i.e `p1`) is of the supertype only and displays the attribute values as of type INTEGER. The second instance (i.e the combination of `p2` and `p_sub`) is of subtype `real_point`. No attributes are shown in the subtype but the values diplayed in the supertype are constrained to be of type REAL.

```
      EXPRESS                          EXPRESS-I
      =======                          =========
ENTITY point;                     p1 = point{x -> 1;
  x : NUMBER;                                y -> 2;};
  y : NUMBER;
END_ENTITY;                       p2 = point{x -> 1.5;
                                             y -> 2.7;
ENTITY real_point                            SUPOF(@p_sub);};
  SUBTYPE OF (point);
  SELF\point.x : REAL;            p_sub = real_point{SUBOF(@p2);};
  SELF\point.y : REAL;
END_ENTITY;
```

In the case where an inherited explicit attribute is redeclared to be a derived attribute, the redeclared attribute shall be treated as a derived attribute in the supertype whenever the redeclaring subtype is instanced.

EXAMPLE 64 – The following *EXPRESS* declares a `circle` to be defined by a centre point and a radius. A `circle_2pt` is a kind of `circle` which is defined by its centre point and a point on the circumference of the circle. The inherited `radius` attribute ios redeclared to be a derived attribute whose value is given by the distance between the two points.

```
ENTITY circle;
  centre : point;
  radius : REAL;
END_ENTITY;

ENTITY circle_2pt
  SUBTYPE OF (circle);
  circum_pnt : point;
DERIVE
  SELF\circle.radius : REAL := distance(SELF\circle.center, circum_pnt);
END_ENTITY;
```

In *EXPRESS-I* instances of `circle` and `circle_2pt` could be:

```
c1 = circle{centre -> [1.0, 0.0];
            radius -> 2.0;};
```

```
c_sup = circle{centre -> [1.0, 0.0];
               radius <- 2.0;
               SUPOF(@c2);};

c2 = circle_2pt{SUBOF(@c_sup);
                circum_pnt -> [1.0, 2.0];};
```

# Annex A
(normative)
# Syntax description of EXPRESS-I

This annex defines the lexical elements of the language and the grammar rules which these elements shall obey.

NOTES

1 – Many of the elements of the *EXPRESS* language are available for use in the definition of test cases. Those elements of *EXPRESS* that are not available are related to the definition of *EXPRESS* schemas, schema interfacing, and rules. For the convenience of the reader, the *EXPRESS* elements are provided here in informative notes. For completeness, the rules relating to the elements of *EXPRESS* that are not available have been provided in the form of comments.

2 – As a further guide, productions which pertain to *EXPRESS-I* only do not use underscores — each name in an *EXPRESS-I* production starts with an upper case letter. For example `DerivedAttr` would be an *EXPRESS-I* production while `derived_attr` would be an *EXPRESS* production. Also, the original numbering of the *EXPRESS* rules has been left intact. The *EXPRESS-I* specific rules have been numbered with an appended 'i'.

3 – This syntax definition will result in ambiguous parsers if taken literally. It has been written to convey information regarding the use of identifiers. The interpreted identifiers define tokens which are references to declared identifiers, and therefore should not resolve to `simple_id`. This requires a parser developer to provide a lookup table, or similar, to enable identifier reference resolution and return the required reference token to a grammar rule checker. This approach has been used to aid the implementors of parsers in that there should be no ambiguity with respect to the use of identifiers.

## A.1 Tokens

The following rules specify the tokens used in *EXPRESS-I*. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses: A.1.1, A.1.2, A.2 and A.3.

### A.1.1 Keywords

This subclause gives the rules used to represent the keywords of *EXPRESS-I*.

NOTE – This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left-hand side is that keyword in uppercase. Since string literals in the syntax rules are case-insensitive, these keywords may be written in *EXPRESS-I* source in upper, lower or mixed case.

```
0i CALL = 'call' .
1i CRITERIA = 'criteria' .
2i END_CALL = 'end_call' .
3i END_CRITERIA = 'end_criteria' .
4i END_NOTES = 'end_notes' .
```

```
5i END_OBJECTIVE = 'end_objective' .
6i END_PARAMETER = 'end_parameter' .
7i END_PURPOSE = 'end_purpose' .
8i END_REALIZATION = 'end_realization' .
9i END_REFERENCES = 'end_references' .

10i END_SCHEMA_DATA = 'end_schema_data' .
11i END_TEST_CASE = 'end_test_case' .
12i IMPORT = 'import' .
13i NOTES = 'notes' .
14i OBJECTIVE = 'objective' .
15i PARAMETER = 'parameter' .
16i PURPOSE = 'purpose' .
17i REALIZATION = 'realization' .
18i REFERENCES = 'references' .
19i SCHEMA_DATA = 'schema_data' .

20i SUBOF = 'subof' .
21i SUPOF = 'supof' .
22i TEST_CASE = 'test_case' .
23i USING = 'using' .
24i WITH = 'with' .
```

NOTE – The following *EXPRESS* rules, numbered 0 through 118 with the exceptions of numbers 8, 37, 38, 49, 84, 89, 90 and 110, are used by *EXPRESS-I*.

```
 0 ABS = 'abs' .
 1 ABSTRACT = 'abstract' .
 2 ACOS = 'acos' .
 3 AGGREGATE = 'aggregate' .
 4 ALIAS = 'alias' .
 5 AND = 'and' .
 6 ANDOR = 'andor' .
 7 ARRAY = 'array' .
 < 8 AS = 'as' . >
 9 ASIN = 'asin' .

10 ATAN = 'atan' .
11 BAG = 'bag' .
12 BEGIN = 'begin' .
13 BINARY = 'binary' .
14 BLENGTH = 'blength' .
15 BOOLEAN = 'boolean' .
16 BY = 'by' .
17 CASE = 'case' .
18 CONSTANT = 'constant' .
19 CONST_E = 'const_e' .

20 CONTEXT = 'context' .
21 COS = 'cos' .
22 DERIVE = 'derive' .
23 DIV = 'div' .
24 ELSE = 'else' .
25 END = 'end' .
26 END_ALIAS = 'end_alias' .
```

```
27 END_CASE = 'end_case' .
28 END_CONSTANT = 'end_constant' .
29 END_CONTEXT = 'end_context' .

30 END_ENTITY = 'end_entity' .
31 END_FUNCTION = 'end_function' .
32 END_IF = 'end_if' .
33 END_LOCAL = 'end_local' .
34 END_MODEL = 'end_model' .
35 END_PROCEDURE = 'end_procedure' .
36 END_REPEAT = 'end_repeat' .
< 37 END_RULE = 'end_rule' . >
< 38 END_SCHEMA = 'end_schema' . >
39 END_TYPE = 'end_type' .

40 ENTITY = 'entity' .
41 ENUMERATION = 'enumeration' .
42 ESCAPE = 'escape' .
43 EXISTS = 'exists' .
44 EXP = 'exp' .
45 FALSE = 'false' .
46 FIXED = 'fixed' .
47 FOR = 'for' .
48 FORMAT = 'format' .
< 49 FROM = 'from' . >

50 FUNCTION = 'function' .
51 GENERIC = 'generic' .
52 HIBOUND = 'hibound' .
53 HIINDEX = 'hiindex' .
54 IF = 'if' .
55 IN = 'in' .
56 INSERT = 'insert' .
57 INTEGER = 'integer' .
58 INVERSE = 'inverse' .
59 LENGTH = 'length' .

60 LIKE = 'like' .
61 LIST = 'list' .
62 LOBOUND = 'lobound' .
63 LOINDEX = 'loindex' .
64 LOCAL = 'local' .
65 LOG = 'log' .
66 LOG10 = 'log10' .
67 LOG2 = 'log2' .
68 LOGICAL = 'logical' .
69 MOD = 'mod' .

70 MODEL = 'model' .
71 NOT = 'not' .
72 NUMBER = 'number' .
73 NVL = 'nvl' .
74 ODD = 'odd' .
75 OF = 'of' .
76 ONEOF = 'oneof' .
```

```
 77 OPTIONAL = 'optional' .
 78 OR = 'or' .
 79 OTHERWISE = 'otherwise' .

 80 PI = 'pi' .
 81 PROCEDURE = 'procedure' .
 82 QUERY = 'query' .
 83 REAL = 'real' .
< 84 REFERENCE = 'reference' . >
 85 REMOVE = 'remove' .
 86 REPEAT = 'repeat' .
 87 RETURN = 'return' .
 88 ROLESOF = 'rolesof' .
< 89 RULE = 'rule . >

< 90 SCHEMA = 'schema' . >
 91 SELECT = 'select' .
 92 SELF = 'self' .
 93 SET = 'set' .
 94 SIN = 'sin' .
 95 SIZEOF = 'sizeof' .
 96 SKIP = 'skip' .
 97 SQRT = 'sqrt' .
 98 STRING = 'string' .
 99 SUBTYPE = 'subtype' .

100 SUPERTYPE = 'supertype' .
101 TAN = 'tan' .
102 THEN = 'then' .
103 TO = 'to' .
104 TRUE = 'true' .
105 TYPE = 'type' .
106 TYPEOF = 'typeof' .
107 UNIQUE = 'unique' .
108 UNKNOWN = 'unknown' .
109 UNTIL = 'until' .

< 110 USE = 'use' . >
111 USEDIN = 'usedin' .
112 VALUE = 'value' .
113 VALUE_IN = 'value_in' .
114 VALUE_UNIQUE = 'value_unique' .
115 VAR = 'var' .
116 WHERE = 'where' .
117 WHILE = 'while' .
118 XOR = 'xor' .
```

### A.1.2 Character classes

The following rules define various classes of characters which are used in constructing the tokens in A.2.

NOTE – The following *EXPRESS* rules, numbered 119 through 135, are used by *EXPRESS-I*.

```
119 bit = '0' | '1' .

120 digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
121 digits = digit { digit } .
122 encoded_character = octet octet octet octet .
123 hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
124 letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' |
             'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
             'w' | 'x' | 'y' | 'z' .
125 lparen_not_star = '(' not_star .
126 not_lparen_star = not_paren_star | ')' .
127 not_paren_star = letter | digit | not_paren_star_special .
128 not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' |
                   ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' |
                   '@' | '[' | '\' | ']' | '^' | '_' | '‘' | '{' | '|' | '}' |
                   '~' .
129 not_paren_star_special = not_paren_star_quote_special | '''' .

130 not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .
131 not_rparen = not_paren_star | '*' | '(' .
132 not_star = not_paren_star | '(' | ')' .
133 octet = hex_digit hex_digit .
134 special = not_paren_star_quote_special | '(' | ')' | '*' | '''' .
135 star_not_rparen = '*' not_rparen .
```

## A.2 Lexical elements

The following rules specify how certain combinations of characters are interpreted as lexical elements within the language.

```
25i BinaryValue = binary_literal .
26i Description = { \a | \s | \n } .
27i EncodedStringValue = '"' { encoded_character | \n } '"' .
28i EnumerationValue = '!' simple_id .
29i IntegerValue = [ sign ] integer_literal .

30i Nil = '?' .
31i SignedMathConstant = [ sign ] MathConstant .
32i SignedRealLiteral = [ sign ] real_literal .
33i SimpleStringValue =  \q { ( \q \q ) | not_quote | \s | \o | \n } \q .
```

NOTE – The following *EXPRESS* rules, numbered 136 through 141, are used by *EXPRESS-I*.

```
136 binary_literal = '%' bit { bit } .
137 encoded_string_literal = '"' encoded_character { encoded_character } '"' .
138 integer_literal = digits .
139 real_literal = digits '.' [ digits ] [ 'e' [ sign ] digits ] .

140 simple_id = letter { letter | digit | '_' } .
141 simple_string_literal = \q { ( \q \q ) | not_quote | \s | \o } \q .
```

### A.2.1 Remarks

The following rules specify the syntax of remarks in *EXPRESS-I*.

NOTE – The following *EXPRESS* rules, numbered 142 through 144, are used by *EXPRESS-I*.

```
142 embedded_remark = '(*' { not_lparen_star | lparen_not_star |
                      star_not_rparen | embedded_remark } '*)' .
143 remark = embedded_remark | tail_remark .
144 tail_remark = '--' { \a | \s | \o } \n .
```

## A.3 Interpreted identifiers

The following rules represent identifiers which are known to have a particular meaning (i.e., to be declared elsewhere as types or functions, etc.).

NOTE – It is expected that identifiers matching these syntax rules are known to an implementation. How the implementation obtains this information is of no concern to the definition of the language. One method of gaining this information is multipass parsing: the first pass collects the identifiers from their declarations, so that subsequent passes are then able to distinguish a `variable_ref` from a `function_ref`, for example.

```
34i ConstantRef = ConstantId .
35i ContextRef = ContextId .
36i EntityInstanceRef = '@' EntityInstanceId .
37i EnumerationInstanceRef = '@' EnumerationInstanceId .
38i ParameterRef = ParameterId .
39i SelectInstanceRef = '@' SelectInstanceId .

40i SimpleInstanceRef = '@' SimpleInstanceId .
41i TypeInstanceRef = '@' TypeInstanceId .
```

NOTE – The following *EXPRESS* rules, numbered 145 through 155, are used by *EXPRESS-I*.

```
145 attribute_ref = attribute_id .
146 constant_ref = constant_id .
147 entity_ref = entity_id .
148 enumeration_ref = enumeration_id .
149 function_ref = function_id .

150 parameter_ref = parameter_id .
151 procedure_ref = procedure_id .
152 schema_ref = schema_id .
153 type_label_ref = type_label_id .
154 type_ref = type_id .
155 variable_ref = variable_id .
```

## A.4 Grammar rules

The following rules specify how the previous lexical elements may be combined into constructs of *EXPRESS-I*. White space and/or remark(s) may appear between any two tokens in these rules. The primary syntax rule for *EXPRESS-I* is `ExpressISyntax`.

```
42i ActualParameter = ParameterRef ':=' ParmValue ';' .
43i AggregationValue = DynamicAggr | FixedAggr .
44i Assignment = variable_id ':=' SelectableInstanceRef ';' .
45i BaseValue = EnumerationValue | SimpleValue .
46i BequeathsTo = SUPOF DynamicEntityRefList ';' .
```

```
47i BooleanValue = TRUE | FALSE .
48i ConstantBlock = CONSTANT { ConstantSpec } END_CONSTANT ';' .
49i ConstantId = constant_ref .

50i ConstantSpec = ConstantId '==' ConstantValue ';' .
51i ConstantValue = AggregationValue | BaseValue | EntityInstanceValue |
                    NamedInstanceValue | SelectValue | TypeValue .
52i ContextBlock = CONTEXT ContextId ';' ContextBody END_CONTEXT ';' .
53i ContextBody = { SchemaReferenceSpec } [ FormalParameterBlock ]
                  { SchemaInstanceBlock | SupportAlgorithm } .
54i ContextId = simple_id .
55i DerattValue = AggregationValue | BaseValue | EntityInstanceRef |
                  EntityInstanceValue | EnumerationInstanceValue |
                  TypeInstanceRef | TypeInstanceValue | TypeValue .
56i DerivedAttr = RoleName [ '<-' DerattValue ] ';' .
57i DynamicAggr = '(' [ DynamicList ] ')' .
58i DynamicEntityRefList = '(' [ EntityRefList ] ')' .
59i DynamicList = DynamicMember { ',' DynamicMember } .

60i DynamicMember = AggregationValue | ConstantValue | DerattValue |
                    ParmValue | ReqattValue | TypeValue .
61i EntityDomain = [ SchemaId '.' ] EntityId .
62i EntityId = entity_ref .
63i EntityInstance = EntityInstanceId '=' EntityInstanceValue ';' .
64i EntityInstanceId = simple_id .
65i EntityInstanceValue = EntityDomain '{'
                          [ InheritsFrom ]
                          { ExplicitAttr }
                          { DerivedAttr }
                          { InverseAttr }
                          [ BequeathsTo ] '}' .
66i EntityRefList = EntityInstanceRef { ',' EntityInstanceRef } .
67i EnumerationDomain = [ SchemaId '.' ] EnumerationId .
68i EnumerationId = type_ref .
69i EnumerationInstance = EnumerationInstanceId '='
                          EnumerationInstanceValue ';' .

70i EnumerationInstanceId = simple_id .
71i EnumerationInstanceValue = EnumerationDomain
                               '{' EnumerationValue '}' .
72i ExplicitAttr = RequiredAttr | OptionalAttr .
73i ExpressISyntax = { TestCaseBlock } { ContextBlock } { ModelBlock }
                     { SchemaInstanceBlock } { ObjectInstance } .
74i FixedAggr = '[' FixedList ']' .
75i FixedList = FixedMember { ',' FixedMember } .
76i FixedMember = DynamicMember | Nil .
77i FormalParameter = ParameterId ':' parameter_type
                      [ ':=' ParmValueDefault ] ';' .
78i FormalParameterBlock = PARAMETER { FormalParameter }
                           END_PARAMETER ';' .
79i ImportSpec = IMPORT '(' { Assignment } ')' ';' .

80i InheritsFrom = SUBOF DynamicEntityRefList ';' .
81i InvattValue = DynamicEntityRefList .
```

```
82i InverseAttr = RoleName [ '<-' InvattValue ] ';' .
83i LogicalValue = logical_literal .
84i MathConstant = CONST_E | PI .
85i ModelBlock = MODEL ModelId ';' ModelBody END_MODEL ';' .
86i ModelBody = { SchemaInstanceBlock } .
87i ModelId = simple_id .
88i NamedInstanceValue = EnumerationInstanceValue | SelectInstanceValue |
                         TypeInstanceValue .
89i NumberValue = IntegerValue | RealValue .

90i ObjectInstance = EntityInstance | EnumerationInstance |
                     SelectInstance | TypeInstance | SimpleInstance .
91i ObjectInstanceRef = EntityInstanceRef | EnumerationInstanceRef |
                        SelectInstanceRef | TypeInstanceRef |
                        SimpleInstanceRef .
92i ObjectiveBlock = OBJECTIVE { TestPurpose } { TestReference }
                     { TestCriteria } { TestNotes } END_OBJECTIVE ';' .
93i OptattValue = ReqattValue | Nil .
94i OptionalAttr = RoleName '->' OptattValue ';' .
95i ParameterId = simple_id .
96i ParameterSpec = WITH '(' { ActualParameter } ')' ';' .
97i ParmValue = ObjectInstanceRef | expression .
98i ParmValueDefault = AggregationValue | BaseValue | ConstantRef |
                        EntityInstanceValue | NamedInstanceValue |
                        ObjectInstanceRef | SelectValue | TypeValue |
                        expression .
99i RealValue = SignedMathConstant | SignedRealLiteral .

100i ReqattValue = AggregationValue | BaseValue | ConstantRef |
                   NamedInstanceValue | ObjectInstanceRef | ParameterRef |
                   SelectValue | TypeValue .
101i RequiredAttr = RoleName '->' ( ReqattValue | Nil ) ';' .
102i RoleName = attribute_ref .
103i SchemaId = schema_ref .
104i SchemaInstanceBlock = SCHEMA_DATA SchemaId ';'
                           [ SchemaInstanceBody ] END_SCHEMA_DATA ';' .
105i SchemaInstanceBody = [ ConstantBlock ] { ObjectInstance } .
106i SchemaReferences = SchemaReferenceSpec { SchemaReferenceSpec } .
107i SchemaReferenceSpec = WITH schema_ref [ USING '(' resource_ref
                           { ',' resource_ref } ')' ] ';' .
108i SelectableInstanceRef = EntityInstanceRef | EnumerationInstanceRef |
                             SelectInstanceRef | TypeInstanceRef .
109i SelectDomain = [ SchemaId  '.' ] SelectId .

110i SelectId = type_ref .
111i SelectInstance = SelectInstanceId '=' SelectInstanceValue ';' .
112i SelectInstanceId = simple_id .
113i SelectInstanceValue = SelectDomain '{' SelectValue '}' .
114i SelectValue = EnumerationValue | NamedInstanceValue |
                  ObjectInstanceRef | TypeValue .
115i SimpleInstance = SimpleInstanceId '=' SimpleValue ';' .
116i SimpleInstanceId = simple_id .
117i SimpleValue = BinaryValue | BooleanValue | LogicalValue |
                   NumberValue | StringValue .
```

```
118i StringValue = SimpleStringValue | EncodedStringValue .
119i SupportAlgorithm = function_decl | procedure_decl .

120i TestCaseBlock = TEST_CASE TestCaseId ';'
                    TestCaseBody END_TEST_CASE ';' .
121i TestCaseBody = SchemaReferences ObjectiveBlock TestRealization
                   { SupportAlgorithm } .
122i TestCaseId = simple_id .
123i TestRealization = REALIZATION { local_decl } { UseContextBlock }
                      { assignment_stmt } END_REALIZATION ';' .
124i TestCriteria = CRITERIA Description END_CRITERIA ';' .
125i TestNotes = NOTES Description END_NOTES ';' .
126i TestPurpose = PURPOSE Description END_PURPOSE ';' .
127i TestReference = REFERENCES Description END_REFERENCES ';' .
128i TypeDomain = [ SchemaId '.' ] TypeId .
129i TypeId = type_ref .

130i TypeInstance = TypeInstanceId '=' TypeInstanceValue ';' .
131i TypeInstanceId = simple_id .
132i TypeInstanceValue = TypeDomain '{' TypeValue '}' .
133i TypeValue = AggregationValue | BaseValue | ConstantRef |
                EntityInstanceValue | NamedInstanceValue |
                ObjectInstanceRef | ParameterRef .
134i UseContextBlock = CALL ContextRef ';'
                      UseContextBody END_CALL ';' .
135i UseContextBody = [ ImportSpec ] [ ParameterSpec ] .
```

NOTE – The following *EXPRESS* grammar rules, numbered 156 through 318 with the exceptions of rules 228, 246, 267, 270, 274, 277–281, 302 and 313, are used by *EXPRESS-I*.

```
156 abstract_supertype_declaration = ABSTRACT SUPERTYPE [ subtype_constraint ] .
157 actual_parameter_list = '(' parameter { ',' parameter } ')' .
158 add_like_op = '+' | '-' | OR | XOR .
159 aggregate_initializer = '[' [ element { ',' element } ] ']' .

160 aggregate_source = simple_expression .
161 aggregate_type = AGGREGATE [ ':' type_label ] OF parameter_type .
162 aggregation_types = array_type | bag_type | list_type | set_type .
163 algorithm_head = { declaration } [ constant_decl ] [ local_decl ] .
164 alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt }
                 END_ALIAS ';' .
165 array_type = ARRAY bound_spec OF [ OPTIONAL ] [ UNIQUE ] base_type .
166 assignment_stmt = general_ref { qualifier } ':=' expression ';' .
167 attribute_decl = attribute_id | qualified_attribute .
168 attribute_id = simple_id .
169 attribute_qualifier = '.' attribute_ref .

170 bag_type = BAG [ bound_spec ] OF base_type .
171 base_type = aggregation_types | simple_types | named_types .
172 binary_type = BINARY [ width_spec ] .
173 boolean_type = BOOLEAN .
174 bound_1 = numeric_expression .
175 bound_2 = numeric_expression .
176 bound_spec = '[' bound_1 ':' bound_2 ']' .
177 built_in_constant = CONST_E | PI | SELF | '?' .
```

```
178 built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS | EXP |
                    FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX |
                    LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF |
                    SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN |
                    VALUE_UNIQUE .
179 built_in_procedure = INSERT | REMOVE .

180 case_action = case_label { ',' case_label } ':' stmt .
181 case_label = expression .
182 case_stmt = CASE selector OF { case_action } [ OTHERWISE ':' stmt ]
            END_CASE ';' .
183 compound_stmt = BEGIN stmt { stmt } END ';' .
184 constant_body = constant_id ':' base_type ':=' expression ';' .
185 constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' .
186 constant_factor = built_in_constant | constant_ref .
187 constant_id = simple_id .
188 constructed_types = enumeration_type | select_type .
189 declaration = entity_decl | function_decl | procedure_decl | type_decl .

190 derived_attr = attribute_decl ':' base_type ':=' expression ';' .
191 derive_clause = DERIVE derived_attr { derived_attr } .
192 domain_rule = [ label ':' ] logical_expression .
193 element = expression [ ':' repetition ] .
194 entity_body = { explicit_attr } [ derive_clause ] [ inverse_clause ]
              [ unique_clause ] [ where_clause ] .
195 entity_constructor = entity_ref '(' [ expression { ',' expression } ] ')' .
196 entity_decl = entity_head entity_body END_ENTITY ';' .
197 entity_head = ENTITY entity_id [ subsuper ] ';' .
198 entity_id = simple_id .
199 enumeration_id = simple_id .

200 enumeration_reference = [ type_ref '.' ] enumeration_ref .
201 enumeration_type = ENUMERATION OF '(' enumeration_id { ',' enumeration_id } ')' .
202 escape_stmt = ESCAPE ';' .
203 explicit_attr = attribute_decl { ',' attribute_decl } ':' [ OPTIONAL ]
                base_type ';' .
204 expression = simple_expression [ rel_op_extended simple_expression ] .
205 factor = simple_factor [ '**' simple_factor ] .
206 formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .
207 function_call = ( built_in_function | function_ref ) [ actual_parameter_list ] .
208 function_decl = function_head [ algorithm_head ] stmt { stmt } END_FUNCTION ';' .
209 function_head = FUNCTION function_id [ '(' formal_parameter
                { ';' formal_parameter } ')' ] ':' parameter_type ';' .

210 function_id = simple_id .
211 generalized_types = aggregate_type | general_aggregation_types | generic_type .
212 general_aggregation_types = general_array_type | general_bag_type |
                            general_list_type | general_set_type .
213 general_array_type = ARRAY [ bound_spec ] OF [ OPTIONAL ] [ UNIQUE ]
                     parameter_type .
214 general_bag_type = BAG [ bound_spec ] OF parameter_type .
215 general_list_type = LIST [ bound_spec ] OF [ UNIQUE ] parameter_type .
216 general_ref =  parameter_ref | variable_ref .
217 general_set_type = SET [ bound_spec ] OF parameter_type .
```

```
218 generic_type = GENERIC [ ':' type_label ] .
219 group_qualifier = '\' entity_ref .
220 if_stmt = IF logical_expression THEN stmt { stmt } [ ELSE stmt { stmt } ]
          END_IF ';' .
221 increment = numeric_expression .
222 increment_control = variable_id ':=' bound_1 TO bound_2 [ BY increment ] .
223 index = numeric_expression .
224 index_1 = index .
225 index_2 = index .
226 index_qualifier = '[' index_1 [ ':' index_2 ] ']' .
227 integer_type = INTEGER .
< 228 interface_specification = reference_clause | use_clause . >
229 interval = '{' interval_low interval_op interval_item interval_op
         interval_high '}' .
230 interval_high = simple_expression .
231 interval_item = simple_expression .
232 interval_low = simple_expression .
233 interval_op = '<' | '<=' .
234 inverse_attr = attribute_decl ':' [ ( SET | BAG ) [ bound_spec ] OF ] entity_ref
               FOR attribute_ref ';' .
235 inverse_clause = INVERSE inverse_attr { inverse_attr } .
236 label = simple_id .
237 list_type = LIST [ bound_spec ] OF [ UNIQUE ] base_type .
238 literal = binary_literal | integer_literal | logical_literal | real_literal |
          string_literal .
239 local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' .
240 local_variable = variable_id { ',' variable_id } ':' parameter_type
                 [ ':=' expression ] ';' .
241 logical_expression = expression .
242 logical_literal = FALSE | TRUE | UNKNOWN .
243 logical_type = LOGICAL .
244 multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
245 named_types = entity_ref | type_ref .
< 246 named_type_or_rename = named_types [ AS ( entity_id | type_id ) ] . >
247 null_stmt = ';' .
248 number_type = NUMBER .
249 numeric_expression = simple_expression .
250 one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .
251 parameter = expression .
252 parameter_id = simple_id .
253 parameter_type = generalized_types | named_types | simple_types .
254 population = entity_ref .
255 precision_spec = numeric_expression .
256 primary = literal | ( qualifiable_factor { qualifier } ) .
257 procedure_call_stmt = ( built_in_procedure | procedure_ref )
                      [ actual_parameter_list ] ';' .
258 procedure_decl = procedure_head [ algorithm_head ] { stmt } END_PROCEDURE ';' .
259 procedure_head = PROCEDURE procedure_id [ '(' [ VAR ] formal_parameter
                 { ';' [ VAR ] formal_parameter } ')' ] ';' .
260 procedure_id = simple_id .
```

```
261 qualifiable_factor = attribute_ref | constant_factor | function_call |
                         general_ref | population .
262 qualified_attribute = SELF group_qualifier attribute_qualifier .
263 qualifier = attribute_qualifier | group_qualifier | index_qualifier .
264 query_expression = QUERY '(' variable_id '<*' aggregate_source '|'
                       logical_expression ')' .
265 real_type = REAL [ '(' precision_spec ')' ] .
266 referenced_attribute = attribute_ref | qualified_attribute .
< 267 reference_clause = REFERENCE FROM schema_ref [ '(' resource_or_rename
                         { ',' resource_or_rename } ')' ] ';' . >
268 rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
269 rel_op_extended = rel_op | IN | LIKE .

< 270 rename_id = constant_id | entity_id | function_id | procedure_id |
                  type_id . >
271 repeat_control = [ increment_control ] [ while_control ] [ until_control ] .
272 repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' .
273 repetition = numeric_expression .
< 274 resource_or_rename = resource_ref [ AS rename_id ] . >
275 resource_ref = constant_ref | entity_ref | function_ref | procedure_ref |
                   type_ref .
276 return_stmt = RETURN [ '(' expression ')' ] ';' .
< 277 rule_decl = rule_head [ algorithm_head ] { stmt } where_clause
                  END_RULE ';' . >
< 278 rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')'
                  ';' . >
< 279 rule_id = simple_id . >

< 280 schema_body = { interface_specification } [ constant_decl ]
                  { declaration | rule_decl } . >
< 281 schema_decl = SCHEMA schema_id ';' schema_body END_SCHEMA ';' . >
282 schema_id = simple_id .
283 selector = expression .
284 select_type = SELECT '(' named_types { ',' named_types } ')' .
285 set_type = SET [ bound_spec ] OF base_type .
286 sign = '+' | '-' .
287 simple_expression = term { add_like_op term } .
288 simple_factor = aggregate_initializer | entity_constructor |
                    enumeration_reference | interval |query_expression |
                    ( [ unary_op ] ( '(' expression ')' | primary ) ) .
289 simple_types = binary_type | boolean_type | integer_type | logical_type |
                   number_type | real_type | string_type .

290 skip_stmt = SKIP ';' .
291 stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt |
           if_stmt | null_stmt |procedure_call_stmt | repeat_stmt | return_stmt |
           skip_stmt .
292 string_literal = simple_string_literal | encoded_string_literal .
293 string_type = STRING [ width_spec ] .
294 subsuper = [ supertype_constraint ] [ subtype_declaration ] .
295 subtype_constraint = OF '(' supertype_expression ')' .
296 subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' .
297 supertype_constraint = abstract_supertype_declaration | supertype_rule .
298 supertype_expression = supertype_factor {  ANDOR supertype_factor } .
```

```
299 supertype_factor = supertype_term { AND supertype_term } .

300 supertype_rule = SUPERTYPE subtype_constraint .
301 supertype_term = entity_ref | one_of | '(' supertype_expression ')' .
< 302 syntax = schema_decl { schema_decl } . >
304 term = factor { multiplication_like_op factor } .
305 type_decl = TYPE type_id '=' underlying_type ';' [ where_clause ] END_TYPE ';' .
306 type_id = simple_id .
307 type_label = simple_id | type_label_ref .
308 unary_op = '+' | '-' | NOT .
309 underlying_type = constructed_types | aggregation_types | simple_types |
                  type_ref .

310 unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } .
311 unique_rule = [ label ':' ] referenced_attribute { ',' referenced_attribute } .
312 until_control = UNTIL logical_expression .
< 313 use_clause = USE FROM schema_ref [ '(' named_type_or_rename
                 { ',' named_type_or_rename } ')' ] ';' . >
314 variable_id = simple_id .
315 where_clause = WHERE domain_rule ';' { domain_rule ';' } .
316 while_control = WHILE logical_expression .
317 width = numeric_expression .
318 width_spec = '(' width ')' [ FIXED ] .
```

## A.5 Cross reference listing

The production on the left is used in the productions indicated on the right.

```
 0i CALL                    | 134i
 1i CRITERIA                | 124i
 2i END_CALL                | 134i
 3i END_CRITERIA            | 124i
 4i END_NOTES               | 125i
 5i END_OBJECTIVE           |  92i
 6i END_PARAMETER           |  78i
 7i END_PURPOSE             | 126i
 8i END_REALIZATION         | 123i
 9i END_REFERENCES          | 127i

10i END_SCHEMA_DATA         | 104i
11i END_TEST_CASE           | 120i
12i IMPORT                  |  79i
13i NOTES                   | 125i
14i OBJECTIVE               |  92i
15i PARAMETER               |  78i
16i PURPOSE                 | 126i
17i REALIZATION             | 123i
18i REFERENCES              | 127i
19i SCHEMA_DATA             | 104i

20i SUBOF                   |  80i
21i SUPOF                   |  46i
22i TEST_CASE               | 120i
```

```
23i USING                     | 107i
24i WITH                      |  96i 107i
25i BinaryValue               | 117i
26i Description               | 124i 125i 126i 127i
27i EncodedStringValue        | 118i
28i EnumerationValue          |  45i  71i 114i
29i IntegerValue              |  89i

30i Nil                       |  48i  76i  93i 101i
31i SignedMathConstant        |  99i
32i SignedRealLiteral         |  99i
33i SimpleStringValue         | 118i
34i ConstantRef               |  98i 100i 133i
35i ContextRef                | 134i
36i EntityInstanceRef         |  55i  66i  91i 108i
37i EnumerationInstanceRef    |  91i 108i
38i ParameterRef              |  42i 100i 133i
39i SelectInstanceRef         |  91i 108i

40i SimpleInstanceRef         |  91i
41i TypeInstanceRef           |  55i  91i 108i
42i ActualParameter           |  96i
43i AggregationValue          |  51i  55i  60i  98i 100i 133i
44i Assignment                |  79i
45i BaseValue                 |  51i  55i  98i 100i 133i
46i BequeathsTo               |  65i
47i BooleanValue              | 117i
48i ConstantBlock             | 105i
49i ConstantId                |  34i  50i

50i ConstantSpec              |  48i
51i ConstantValue             |  50i  60i
52i ContextBlock              |  73i
53i ContextBody               |  52i
54i ContextId                 |  35i  52i
55i DerattValue               |  56i  60i
56i DerivedAttr               |  65i
57i DynamicAggr               |  43i
58i DynamicEntityRefList      |  45i  80i  82i
59i DynamicList               |  57i

60i DynamicMember             |  59i  76i
61i EntityDomain              |  65i  88i
62i EntityId                  |  61i
63i EntityInstance            |  90i
64i EntityInstanceId          |  36i  63i
65i EntityInstanceValue       |  51i  55i  63i  98i 133i
66i EntityRefList             |  58i
67i EnumerationDomain         |  71i  88i
68i EnumerationId             |  67i
69i EnumerationInstance       |  90i
```

```
 70i EnumerationInstanceId    |  37i  69i
 71i EnumerationInstanceValue |  55i  69i  88i
 72i ExplicitAttr             |  65i
 73i ExpressISyntax           |
 74i FixedAggr                |  43i
 75i FixedList                |  74i
 76i FixedMember              |  75i
 77i FormalParameter          |  78i
 78i FormalParameterBlock     |  53i
 79i ImportSpec               | 135i

 80i InheritsFrom             |  65i
 81i InvattValue              |  82i
 82i InverseAttr              |  65i
 83i LogicalValue             |  48i 117i
 84i MathConstant             |  48i
 85i ModelBlock               |  73i
 86i ModelBody                |  85i
 87i ModelId                  |  38i  85i
 88i NamedInstanceValue       |  51i  98i 100i 114i 133i
 89i NumberValue              | 117i

 90i ObjectInstance           |  73i 105i
 91i ObjectInstanceRef        |  97i  98i 100i 114i 133i
 92i ObjectiveBlock           | 121i
 93i OptattValue              |  94i
 94i OptionalAttr             |  72i
 95i ParameterId              |  38i  77i
 96i ParameterSpec            | 135i
 97i ParmValue                |  42i  60i
 98i ParmValueDefault         |  77i
 99i RealValue                |  89i

100i ReqattValue              |  60i  93i 101i
101i RequiredAttr             |  72i
102i RoleName                 |  56i  82i  94i 101i
103i SchemaId                 |  61i  67i 104i 109i 128i
104i SchemaInstanceBlock      |  53i  73i  86i
105i SchemaInstanceBody       | 104i
106i SchemaReferences         | 121i
107i SchemaReferenceSpec      |  53i 106i
108i SelectableInstanceRef    |  44i
109i SelectDomain             |  88i 113i

110i SelectId                 | 109i
111i SelectInstance           |  90i
112i SelectInstanceId         |  39i 111i
113i SelectInstanceValue      |  88i 111i
114i SelectValue              |  51i  98i 100i 113i
115i SimpleInstance           |  90i
116i SimpleInstanceId         |  40i 115i
117i SimpleValue              |  45i 115i
```

```
118i StringValue              | 117i
119i SupportAlgorithm         |  53i 121i

120i TestCaseBlock            |  73i
121i TestCaseBody             | 120i
122i TestCaseId               | 120i
123i TestRealization          | 121i
124i TestCriteria             |  92i
125i TestNotes                |  92i
126i TestPurpose              |  92i
127i TestReference            |  92i
128i TypeDomain               |  87i 132i
129i TypeId                   |  128i

130i TypeInstance             |  90i
131i TypeInstanceId           |  41i 130i
132i TypeInstanceValue        |  55i  88i 130i
133i TypeValue                |  51i  55i  60i  98i 100i 114i 132i
134i UseContextBlock          | 123i
135i UseContextBody           | 134i

   0 ABS                      | 178
   1 ABSTRACT                 | 156
   2 ACOS                     | 178
   3 AGGREGATE                | 161
   4 ALIAS                    | 164
   5 AND                      | 244 299
   6 ANDOR                    | 298
   7 ARRAY                    | 165 213
   8
   9 ASIN                     | 178

  10 ATAN                     | 178
  11 BAG                      | 170 214 234
  12 BEGIN                    | 183
  13 BINARY                   | 172
  14 BLENGTH                  | 178
  15 BOOLEAN                  | 173
  16 BY                       | 222
  17 CASE                     | 182
  18 CONSTANT                 | 185 48i
  19 CONST_E                  | 177 84i

  20 CONTEXT                  | 52i
  21 COS                      | 178
  22 DERIVE                   | 191
  23 DIV                      | 244
  24 ELSE                     | 220
  25 END                      | 183
  26 END_ALIAS                | 164
  27 END_CASE                 | 182
  28 END_CONSTANT             | 185 48i
```

```
29 END_CONTEXT             | 52i

30 END_ENTITY              | 196
31 END_FUNCTION            | 208
32 END_IF                  | 220
33 END_LOCAL               | 239
34 END_MODEL               | 85i
35 END_PROCEDURE           | 258
36 END_REPEAT              | 272
37
38
39 END_TYPE                | 304

40 ENTITY                  | 197
41 ENUMERATION             | 201
42 ESCAPE                  | 202
43 EXISTS                  | 178
44 EXP                     | 178
45 FALSE                   | 242 47i
46 FIXED                   | 318
47 FOR                     | 164 234
48 FORMAT                  | 178
49

50 FUNCTION                | 209
51 GENERIC                 | 218
52 HIBOUND                 | 178
53 HIINDEX                 | 178
54 IF                      | 220
55 IN                      | 269
56 INSERT                  | 179
57 INTEGER                 | 227 81i
58 INVERSE                 | 235
59 LENGTH                  | 178

60 LIKE                    | 269
61 LIST                    | 215 237
62 LOBOUND                 | 178
63 LOCAL                   | 239
64 LOG                     | 178
65 LOG10                   | 178
66 LOG2                    | 178
67 LOGICAL                 | 243
68 LOINDEX                 | 178
69 MOD                     | 244

70 MODEL                   | 85i
71 NOT                     | 308
72 NUMBER                  | 248
73 NVL                     | 178
74 ODD                     | 178
75 OF                      | 161 165 170 182 201 213 214 215 217 234 237 285 295
```

```
                               296
 76 ONEOF                    | 250
 77 OPTIONAL                 | 165 203 213
 78 OR                       | 158
 79 OTHERWISE                | 182

 80 PI                       | 177 84i
 81 PROCEDURE                | 259
 82 QUERY                    | 264
 83 REAL                     | 265
 84
 85 REMOVE                   | 179
 86 REPEAT                   | 272
 87 RETURN                   | 276
 88 ROLESOF                  | 178
 89

 90
 91 SELECT                   | 284
 92 SELF                     | 177 262
 93 SET                      | 217 234 285
 94 SIN                      | 178
 95 SIZEOF                   | 178
 96 SKIP                     | 290
 97 SQRT                     | 178
 98 STRING                   | 293
 99 SUBTYPE                  | 296

100 SUPERTYPE                | 156 300
101 TAN                      | 178
102 THEN                     | 220
103 TO                       | 222
104 TRUE                     | 242 47i
105 TYPE                     | 304
106 TYPEOF                   | 178
107 UNIQUE                   | 165 213 215 237 310
108 UNKNOWN                  | 242
109 UNTIL                    | 312

110
111 USEDIN                   | 178
112 VALUE                    | 178
113 VALUE_IN                 | 178
114 VALUE_UNIQUE             | 178
115 VAR                      | 259
116 WHERE                    | 315
117 WHILE                    | 316
118 XOR                      | 158
119 bit                      | 136

120 digit                    | 121 123 127 130 140
121 digits                   | 138 139
```

```
122 encoded_character        | 137 27i
123 hex_digit                | 133
124 letter                   | 127 130 140
125 lparen_not_star          | 142
126 not_lparen_star          | 142
127 not_paren_star           | 126 131 132
128 not_paren_star_quote_special | 129 130 134
129 not_paren_star_special   | 127

130 not_quote                | 141 33i
131 not_rparen               | 135
132 not_star                 | 125
133 octet                    | 122
134 special                  |
135 star_not_rparen          | 142
136 binary_literal           | 238 25i
137 encoded_string_literal   | 292
138 integer_literal          | 238
139 real_literal             | 238

140 simple_id                | 168 187 198 199 210 236 252 260 282 305 307 314
                               28i 54i 64i 70i 87i 95i 112i 116i 122i 131i
141 simple_string_literal    | 292
142 embedded_remark          | 142 143
143 remark                   |
144 tail_remark              | 143
145 attribute_ref            | 169 234 261 266 102i
146 constant_ref             | 186 275 49i
147 entity_ref               | 195 219 234 245 254 275 296 301 62i
148 enumeration_ref          | 200
149 function_ref             | 207 275

150 parameter_ref            | 216
151 procedure_ref            | 257 275
152 schema_ref               | 103i 107i
153 type_label_ref           | 306
154 type_ref                 | 200 245 275 309 68i 110i 129i
155 variable_ref             | 216
156 abstract_supertype_declaration | 297
157 actual_parameter_list    | 207 257
158 add_like_op              | 287
159 aggregate_initializer    | 288

160 aggregate_source         | 264
161 aggregate_type           | 211
162 aggregation_types        | 171 309
163 algorithm_head           | 208 258
164 alias_stmt               | 291
165 array_type               | 162
166 assignment_stmt          | 291 123i
167 attribute_decl           | 190 203 234
168 attribute_id             | 145 167
```

```
169 attribute_qualifier      | 262 263

170 bag_type                 | 162
171 base_type                | 165 170 184 190 203 237 285
172 binary_type              | 289
173 boolean_type             | 289
174 bound_1                  | 176 222
175 bound_2                  | 176 222
176 bound_spec               | 165 170 213 214 215 217 234 237 285
177 built_in_constant        | 186
178 built_in_function        | 207
179 built_in_procedure       | 257

180 case_action              | 182
181 case_label               | 180
182 case_stmt                | 291
183 compound_stmt            | 291
184 constant_body            | 185
185 constant_decl            | 163
186 constant_factor          | 261
187 constant_id              | 146 184
188 constructed_types        | 309
189 declaration              | 163

190 derived_attr             | 191
191 derive_clause            | 194
192 domain_rule              | 315
193 element                  | 159
194 entity_body              | 196
195 entity_constructor       | 288
196 entity_decl              | 189
197 entity_head              | 196
198 entity_id                | 147 197
199 enumeration_id           | 148 201

200 enumeration_reference    | 288
201 enumeration_type         | 188
202 escape_stmt              | 291
203 explicit_attr            | 194
204 expression               | 166 181 184 190 193 195 240 241 251 276 283 288
                               97i 98i
205 factor                   | 303
206 formal_parameter         | 209 259
207 function_call            | 261
208 function_decl            | 189 119i
209 function_head            | 208

210 function_id              | 149 209
211 generalized_types        | 253
212 general_aggregation_types | 211
213 general_array_type       | 212
214 general_bag_type         | 212
```

```
215 general_list_type        | 212
216 general_ref              | 164 166 261
217 general_set_type         | 212
218 generic_type             | 211
219 group_qualifier          | 262 263

220 if_stmt                  | 291
221 increment                | 222
222 increment_control        | 271
223 index                    | 224 225
224 index_1                  | 226
225 index_2                  | 226
226 index_qualifier          | 263
227 integer_type             | 289
228
229 interval                 | 288

230 interval_high            | 229
231 interval_item            | 229
232 interval_low             | 229
233 interval_op              | 229
234 inverse_attr             | 235
235 inverse_clause           | 194
236 label                    | 192 311
237 list_type                | 162
238 literal                  | 256
239 local_decl               | 163 123i

240 local_variable           | 239
241 logical_expression       | 192 220 264 312 316
242 logical_literal          | 238 83i
243 logical_type             | 289
244 multiplication_like_op   | 303
245 named_types              | 171 253 284
246
247 null_stmt                | 291
248 number_type              | 289
249 numeric_expression       | 174 175 221 223 255 273 317

250 one_of                   | 301
251 parameter                | 157
252 parameter_id             | 150 206
253 parameter_type           | 161 206 209 213 214 215 217 240 77i
254 population               | 261
255 precision_spec           | 265
256 primary                  | 288
257 procedure_call_stmt      | 291
258 procedure_decl           | 189 119i
259 procedure_head           | 258

260 procedure_id             | 151 259
261 qualifiable_factor       | 256
```

```
262 qualified_attribute      | 167 266
263 qualifier                | 164 166 256
264 query_expression         | 288
265 real_type                | 289
266 referenced_attribute     | 311
267
268 rel_op                   | 269
269 rel_op_extended          | 204

270
271 repeat_control           | 272
272 repeat_stmt              | 291
273 repetition               | 193
274
275 resource_ref             | 107i
276 return_stmt              | 291
277
278
279

280
281
282 schema_id                | 152
283 selector                 | 182
284 select_type              | 188
285 set_type                 | 162
286 sign                     | 139 29i 31i 32i
287 simple_expression        | 160 204 230 231 232 249
288 simple_factor            | 205
289 simple_types             | 171 253 309

290 skip_stmt                | 291
291 stmt                     | 164 180 182 183 208 220 258 272
292 string_literal           | 238
293 string_type              | 289
294 subsuper                 | 197
295 subtype_constraint       | 156 300
296 subtype_declaration      | 294
297 supertype_constraint     | 294
298 supertype_expression     | 250 295 301
299 supertype_factor         | 298

300 supertype_rule           | 297
301 supertype_term           | 299
302
303 term                     | 287
304 type_decl                | 189
305 type_id                  | 154 304
306 type_label               | 161 218
307 type_label_id            | 153 306
308 unary_op                 | 288
309 underlying_type          | 304
```

```
310 unique_clause            | 194
311 unique_rule              | 310
312 until_control            | 271
313
314 variable_id              | 155 164 222 240 264 44i
315 where_clause             | 194 304
316 while_control            | 271
317 width                    | 318
318 width_spec               | 172 293
```

# Annex B
(normative)

# Protocol implementation conformance statement (PICS)

Is this implementation an *EXPRESS-I* language parser/verifier? If so, answer the questions provided in B.1.

## B.1 EXPRESS-I language parser

For which level is support claimed:

- ☐ Level 1 – Reference checking;
- ☐ Level 2 – Type checking;
- ☐ Level 3 – Value checking;
- ☐ Level 4 – Complete checking.

(Note: In order to claim support for a given level, all lower levels must also be supported.)

| | |
|---|---|
| What is the maximum integer value [integer_literal]?: | ...........................: |
| What is the maximum real precision [real_literal]?: | ...........................: |
| What is the maximum real exponent [real_literal]?: | ...........................: |
| What is the maximum string width (characters) [simple_string_literal]?: | ...........................: |
| What is the maximum string width (octets) [encoded_string_literal]?: | ...........................: |
| What is the maximum binary width (bits) [binary_literal]?: | ...........................: |
| Do you have a limit on the number of unique identifiers which are declared? If so, what is your limit?: | ...........................: |
| Do you have a limit on the number of characters used as an identifier? If so, what is your limit?: | ...........................: |
| Do you have a limit on the scope nesting depth? If so, what is your limit?: | ...........................: |
| How do you represent the standard constant '?' [built_in_constant]?: | ...........................: |

# Annex C
(normative)
# Information object registration

In order to provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(12) version(1) }

is assigned to this part of ISO 10303. The meaning of this value is defined in ISO 8824-1, and is described in ISO 10303-1.

# Annex D
(informative)

# Language specification syntax

The notation used to present the syntax of the *EXPRESS-I* language is defined in ISO 10303-11. It is repeated here for informational purposes.

The full syntax for the *EXPRESS-I* language is given in normative annex A. Portions of those syntax rules are reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not always complete so it will sometimes be necessary to consult annex A for the missing rules. The syntax portions within this International Standard are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross references to other syntax rules.

## D.1 The syntax of the specification

The syntax of *EXPRESS* (and *EXPRESS-I* ) is defined in a derivative of Wirth Syntax Notation (WSN); see annex G under [2] for a reference.

The notational conventions and WSN defined in itself are given below.

```
syntax          = { production } .
production      = identifier '=' expression '.' .
expression      = term { '|' term } .
term            = factor { factor } .
factor          = identifier | literal | group | option | repetition .
identifier      = character { character } .
literal         = '''' character { character } '''' .
group           = '(' expression ')' .
option          = '[' expression ']' .
repetition      = '{' expression '}' .
```

– The equal sign '=' indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period '.'.

– The use of an identifier within a factor denotes a nonterminal symbol which appears on the left side of another production. An identifier is composed of letters, digits and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol which cannot be expanded further. A literal is a case independent sequence of characters enclosed in apostrophes. Character, in this case, stands for any character as defined by ISO 10646 cells 21-7E in group 00, plane 00, row 00. For an apostrophe to appear in a literal it must be written twice.

– The semantics of the enclosing braces are defined below:

  - curly braces '{ }' indicates zero or more repetitions;

  - square brackets '[ ]' indicates optional parameters;

  - parenthesis '()' indicates that the group of productions enclosed by parenthesis shall be used as a single production;

  - vertical bar '|' indicates that exactly one of the terms in the expression shall be chosen.

NOTE – For the purposes of this document, one further construct has been added the the meta-language above. A comment is any text enclosed within angle brackets. For example, `< A comment >` is a comment.

EXAMPLE 65 – The syntax for a real literal is as follows:

**Syntax:**

```
190 real_literal = integer_literal '.' [ integer_literal ]
                     [ 'e' [ sign ] integer_literal ] .
163 integer_literal = digit { digit } .
```

The complete syntax definition (annex A) contains the definitions for `sign` and `digit`.

EXAMPLE 66 – Following the syntax given in example 65, the following alternatives are possible:

a) `123.`

b) `123.456`

c) `123.456e7`

d) `123.456E-7`

## D.2 Special character notation

The following notation is used to represent entire character sets and certain special characters which are difficult to display.

– `\a` represents characters in cells 21-7E of row 00, plane 00, group 00 of ISO 10646;

– `\n` represents a newline (system dependent);

– `\q` is the quote (apostrophe) (') character and is contained within `\a`;

– `\s` is the space character;

– `\o` represents characters in cells 00-1F and 7F of row 00, plane 00, group 00 of ISO 10646.

# Annex E
(informative)
# Example test cases

This annex provides some examples of test cases. These examples are not intended to be indicative of any normative test cases that may be given in other parts of this International Standard and are given purely for illustrative purposes.

First we start with a simple *EXPRESS* SCHEMA against which the test cases are specified.

```
*)
SCHEMA people;

  TYPE name : STRING; END_TYPE;

  ENTITY person;
    named : name;
    children : SET [0:?] OF person;
  END_ENTITY;

  ENTITY male
    SUBTYPE OF (person);
  END_ENTITY;

  ENTITY female
    SUBTYPE OF (person);
  END_ENTITY;

  ENTITY married;
    husband : male;
    wife    : female;
  END_ENTITY;

END_SCHEMA;
(*
```

## E.1 Test case 1

This test case specifies that three instances of `person` are to be created.

```
*)
TEST_CASE test_case_1;

  WITH people USING(person);

  OBJECTIVE
    PURPOSE To test the creation of supertypes with no subtypes. END_PURPOSE;
    REFERENCES None. END_REFERENCES;
    CRITERIA Three instances of childless PERSON shall be created. END_CRITERIA;
    NOTES None. END_NOTES;
  END_OBJECTIVE;
  REALIZATION

    LOCAL               -- define variables of type person
      p1 : person;
      p2 : person;
      p3 : person;
    END_LOCAL;

    p1 := person('Alpha', []);  -- create instances of person
    p2 := person('Beta', []);
    p3 := person('Gamma', []);

  END_REALIZATION;

END_TEST_CASE;
(*
```

One possible rendition of the data resulting from this test case is:

```
*)
MODEL case_1;
  SCHEMA_DATA people;

  n1 = name{'Alpha'};
  n2 = name{'Beta'};
  n3 = name{'Gamma'};

  p1 = person{named    -> @n1;
              children -> ();
              SUPOF();};

  p2 = person{named    -> @n2;
              children -> ();
              SUPOF();};

  p3 = person{named    -> @n3;
              children -> ();
              SUPOF();};

  END_SCHEMA_DATA;
END_MODEL;
(*
```

For future use, the following context is defined, based on the test case.

```
*)
CONTEXT context_1;
  SCHEMA_DATA people;

  p1 = person{named    -> 'Alpha';
              children -> ();
              SUPOF();};
  p2 = person{named    -> 'Beta';
              children -> ();
              SUPOF();};

  p3 = person{named    -> 'Gamma';
              children -> ();
              SUPOF();};

  END_SCHEMA_DATA;
END_CONTEXT;
(*
```

## E.2 Test case 2

This test case creates a `male` and `female` person.

```
*)
TEST_CASE test_case_2;

  WITH people USING(male, female);

  OBJECTIVE
    PURPOSE  To test the creation of subtypes. END_PURPOSE;
    CRITERIA One instance of a childless MAN and one of a childless
             FEMALE shall be created. END_CRITERIA;
  END_OBJECTIVE;

  REALIZATION

    LOCAL                   -- define variables of the required types
      m1 : male;
      f1 : female;
    END_LOCAL;

    m1 := person('Adam', [])||male();    -- create male instance
    f1 := person('Eve'), [])||female();  -- create female instance

  END_REALIZATION;

END_TEST_CASE;
(*
```

One possible rendition of the data resulting from this test case is:

```
*)
MODEL case_2;
  SCHEMA_DATA people;

  p4 = person{named    -> 'Adam';
              children -> ();
              SUPOF(@m1);};

  m1 = male{SUBOF(@p4);};
  p5 = person{named    -> 'Eve';
              children -> ();
              SUPOF(@f1);};

  f1 = female{SUBOF(@p5);};

  END_SCHEMA_DATA;
END_MODEL;
(*
```

For future use, the following parameterised context is also created.

```
*)
CONTEXT context_2;

  WITH people USING(person);

  PARAMETER
    c1 : SET OF person := ();    -- parameter default is the empty set
    c2 : SET OF person := ();
  END_PARAMETER;

  SCHEMA_DATA people;

  p4 = person{named    -> 'Adam';
              children -> c1;      -- children attribute is parameterised
              SUPOF(@m1);};

  m1 = male{SUBOF(@p4);};

  p5 = person{named    -> 'Eve';
              children -> c2;
              SUPOF(@f1);};

  f1 = female{SUBOF(@p5);};

  END_SCHEMA_DATA;
END_CONTEXT;
(*
```

## E.3 Test case 3

This test creates an instance of a `married` entity.

```
*)
TEST_CASE test_case_3;

  WITH people USING(married);

  OBJECTIVE
    PURPOSE  To test the creation of an entity with attributes
             of type entity. END_PURPOSE;
    CRITERIA One instance of a MARRIED entity shall be created. END_CRITERIA;
  END_OBJECTIVE;

  REALIZATION

    LOCAL                     -- define variables of required types
      reg : married;
      h1  : male;
      w1  : female;
    END_LOCAL;

    CALL context_2;             -- use data from CONTEXT context_2
      IMPORT(h1 := @m1;
             w1 := @f1;);
    END_CALL;

    reg := married(h1, w1);   -- create instance of married

  END_REALIZATION;

END_TEST_CASE;
(*
```

One possible rendition of the data resulting from this test case is:

```
*)
MODEL case_3;
  SCHEMA_DATA people;

  p4 = person{named    -> 'Adam';
              children -> ();
              SUPOF(@h1);};

  h1 = male{SUBOF(@p4);};

  p5 = person{named    -> 'Eve';
              children -> ();
              SUPOF(@w1);};

  w1 = female{SUBOF(@p5);};

  reg = married{husband -> @h1;
                wife    -> @w1;};

  END_SCHEMA_DATA;
END_MODEL;
(*
```

## E.4 Test case 4

This test case assembles a set of pre-existing parameterised data and also creates new data.

```
*)
```

```
TEST_CASE test_case_4;

  WITH people USING(person, male, female, married);

  OBJECTIVE
    PURPOSE  To test the creation of a married couple with
             children. END_PURPOSE;
    CRITERIA Three instances of PERSON shall be created.
             One instance each of MALE and FEMALE with children shall
             be created.
             One instance of a MARRIED entity shall be created.
    END_CRITERIA;
  END_OBJECTIVE;

  REALIZATION

    LOCAL               -- define variables of the required types
      p1 : person;
      p2 : person;
      p3 : person;
      m1 : male;
      f1 : female;
      reg : married;
    END_LOCAL;

    CALL context_1;
      IMPORT(p1 := @p1;         -- use data from CONTEXT context_1
             p2 := @p2;
             p3 := @p3;);
    END_CALL;

    CALL context_2;
      IMPORT(m1 := @m1;         -- use data from CONTEXT context_2
             f1 := @f1;);
      WITH(c1 := [p1, p3];      -- set parameter values
           c2 := [p2, p3];);
    END_CALL;

    reg := married(m1, f1);     -- create married instance

  END_REALIZATION;

END_TEST_CASE;
(*
```

One possible rendition of the data resulting from this test case is:

```
*)
MODEL case_4;
  SCHEMA_DATA people;

  n1 = name{'Alpha'};
  n2 = name{'Beta'};
  n3 = name{'Gamma'};

  p1 = person{named    -> @n1;
              children -> ();
              SUPOF();};

  p2 = person{named    -> @n2;
              children -> ();
              SUPOF();};

  p3 = person{named    -> @n3;
              children -> ();
              SUPOF();};

  p4 = person{named    -> 'Adam';
              children -> (@p1, @p3);
              SUPOF(@m1);};

  m1 = male{SUBOF(@p4);};

  p5 = person{named    -> 'Eve';
              children -> (@p2, @p3);
              SUPOF(@f1);};

  f1 = female{SUBOF(@p5);};

  reg = married{husband -> @m1;
                wife    -> @f1;};

  END_SCHEMA_DATA;
END_MODEL;
(*
```

# Annex F
(informative)
# Usage notes

This annex discusses some of the potential uses of the *EXPRESS-I* language.

In Object-Oriented terms, an *EXPRESS* entity would be called a *class*, and an instance of a class is termed an *object*; one object may reference another object. *EXPRESS* distinguishes between entities and types (i.e the ENUMERATION, SELECT and the defined data TYPE) as entities may be subtyped whereas types cannot be subtyped. The physical file, as defined in ISO 10303 Part 21, certainly distinguishes between entities and types in that only entity instances may appear in the file — type values are embedded within the attribute values and are not referenceable. *EXPRESS-I* treats entity instances as objects in the OO sense. It also allows types to be treated as objects, in that they can be instantiated and referenced; alternatively, it allows types to be treated in the same manner as in the physical file in that their values can be embedded.

## F.1 EXPRESS data examples

The simplest use of *EXPRESS-I* is as a paper exercise in displaying data populated examples of *EXPRESS* defined constructs. The language allows the display of entity instances as referenceable objects. Types instances may also be displayed as referenceable objects, or they may appear as unreferenceable values within other objects' values. Examples given in this document show both forms of type instantiation.

Values of explicit entity attributes are required. The values of derived or inverse attributes need not be displayed, except as exemplars, because as noted, these are essentially calculable from the values of the explicit attributes.

Examples of *EXPRESS* schemas can also be displayed, as well as individual objects.

The *EXPRESS-I* MODEL construct is provided to enable the display of multiple schemas. Typically, a MODEL would be used when two or more *EXPRESS* schemas interact with each other. Note that *EXPRESS* itself does not support such a construct.

## F.2 Abstract test cases

The *EXPRESS-I* TEST_CASE construct is provided to assist in the formal specification of test cases against the implementation of *EXPRESS* defined constructs. *EXPRESS* itself does not provide an equivalent construct.

For a test case, a base set of *EXPRESS-I* objects must be defined which will be those objects, and their supporting data, to be tested. The values of these objects may be in the form of parameters, whose formal definition are given in an enclosing CONTEXT. A series of test cases may then be defined on the CONTEXT, by providing actual parameter values. Thus, a single "parameterized" context may support many different tests. The test case documentation will also have to include the test purposes and expected results.

## F.3 Object bases

Here, we assume the availabilty of some object base that stores objects according to *EXPRESS* defined schema(s). That is, the object base has the capability of maintaining a partitioning of the objects according to the *EXPRESS* schemas in which their definitions are declared. The design and implementation of such an object base is left as an exercise for the reader.

### F.3.1 Input

Given an object base, *EXPRESS-I* could be used as one means of inputting objects into the object base. This process could be either a batch process, where a previously prepared *EXPRESS-I* file was read into the object processor, or it could be an interactive process, where the user incrementally added *EXPRESS-I* objects.

Depending on the sophistication of the object base, the user may or may not need to explicitly provide values for derived and inverse attributes.

### F.3.2 Output

Given a populated object base, *EXPRESS-I* could be used as a data output language for displaying some or all of the contents of the object base to a human reader.

Depending on the sophistication of the object base, the displayed entity objects may or may not include values for derived and inverse attributes. Note, though, that at least the role names of these attributes are required.

The *EXPRESS-I* MODEL construct is designed for the display of the population of an object base.

### F.3.3 Code testing

Ideally, an implementation of an object base should provide functionality to evaluate all the constraints on the *EXPRESS* entities and types that may occur as objects or values within the object base. For instance, an *EXPRESS* schema may contain an ENTITY definition that includes a derived attribute and a constraint on the derived value. An object base should be able to both evaluate the derived attribute and also reject any object of that ENTITY class whose attribute values do not satisfy the constraints. This requires code. *EXPRESS-I* could be used as data input for testing such code.

Other code examples include:

– Determination of the values of inverse attributes.

– Checking uniqeness constraints across an object population.

– Code to implement *EXPRESS* defined RULEs.

Note that these types of functions are also required for physical file test systems and other forms of exchange data processors.

## F.4 Non-EXPRESS data examples

As *EXPRESS-I* entity instances are in the form of named tuples, it may also be used to display objects or records from languages other than *EXPRESS*. For example, instances of C `structs` or the state of objects representing instances of classes from Object Oriented languages such as C++ or Eiffel. Similarly for languages that support Frames.

EXAMPLE 67 – A C language `struct` may be defined as:

```
struct point {
    int x;
    int y;
};
```

An *EXPRESS-I* instance of this `struct` could appear as:

```
p1 = point{x -> 10;
           y -> 20;};
```

The language may be used to represent tabular data from relational databases, where the entity name is equivalent to a table name, and each instance is a (identified) line in the table, or network or Object Oriented type databases. In another vein it could be used as a file format-independent representation for IGES data.

EXAMPLE 68 – A table in a relational database may be defined by the following SQL:

```
CREATE TABLE PART
    ( ID      CHAR(6)  NOT NULL;
      PNAME   CHAR(20) NOT NULL;
      COLOR   CHAR(6)  NOT NULL;
      WEIGHT  SMALLINT NOT NULL;
      CITY    CHAR(15) NOT NULL;
    PRIMARY KEY ( ID ) ;
```

Instances of two of the rows from a populated `PART` table could be represented by *EXPRESS-I* as:

```
part_row1 = PART{ID -> 'p33';
                 PNAME -> 'Nut';
                 COLOR -> 'Red';
                 WEIGHT -> 12;
                 CITY -> 'Paris'; };
part_row2 = PART{ID -> 'p8';
                 PNAME -> 'Washer';
                 COLOR -> 'Green';
                 WEIGHT -> 4;
                 CITY -> 'Rome'; };
```

An example of a completely different usage is given by Godwin *et al* [3] who have proposed *EXPRESS-I* as being the formal meta language for the Semantic Unification Meta Model [4], which in turn is based on predicate logic.

# Annex G
(informative)
# Bibliography

1. ISO TR 9007; *"Information processing systems - Concepts and terminology for the conceptual schema and information base",* 1987.

2. WIRTH, N.; *"What can we do about the unnecessary diversity of notation for syntactic definitions?",* Communications of the ACM, November 1977, vol 20, no. 11, p. 822.

3. GODWIN, A.N., GIANNASI, F. and TAHZIB, S.; *"An example using the SUMM with EXPRESS and relational models",* in WILSON, P.R. (editor) *EUG'94: 4th Annual EXPRESS User Group International Conference,* Greenville, SC, 13–14 October, 1994.

4. FULTON, J.A. et al; *"Technical report on the Semantic Unification Meta-Model: Volume 1 — Semantic unification of static models",* ISO TC184/SC4 WG3 Document N175, October 1992.

# Index